

msdn magazine



Migrate Your JavaScript
Code to TypeScript.....26

Zero to Dashboard in Record Time.

DevExpress Dashboard is the right tool for business because it delivers flexible, interactive and fully customizable user experiences so you can create enterprise-ready decision support systems in the shortest possible time.

Get started today: DevExpress.com/Dashboard





When Only the Best Will Do.

High-Performance, Elegant and Feature-Complete
.NET Controls and Libraries.

Download your free 30-day trial today and Experience the DevExpress Difference.

www.devexpress.com/try

msdn

magazine



Migrate Your JavaScript
Code to TypeScript.....26

Enhance Your JavaScript Investment with TypeScript Bill Wagner	26
Override the Default Scaffold Templates Jonathan Waldman	34
Topshelf and Katana: A Unified Web and Service Architecture Wes McClure	46
The MVVM Light Messenger In-Depth Laurent Bugnion	54
Using JSLink with SharePoint 2013 Pritam Baldota	62

COLUMNS

CUTTING EDGE

External Authentication
with ASP.NET Identity
Dino Esposito, page 6

WINDOWS WITH C++

High-Performance Window
Layering Using the Windows
Composition Engine
Kenny Kerr, page 14

AZURE INSIDER

Telemetry Ingestion
and Analysis Using
Microsoft Azure Services
Thomas Conte,
Bruno Terkaly,
Ricardo Villalobos, page 20

TEST RUN

Working with the MNIST Image
Recognition Data Set
James McCaffrey, page 68

DIRECTX FACTOR

The Canvas and the Camera
Charles Petzold, page 74

DON'T GET ME STARTED

VB6 and the Art
of the Knuckleball
David Platt, page 80

#1

**1: As a Scrum team, I want to
get more \$#*% done**

Assigned To: Your Team

Priority: Very High

Release: V 1.0

0 sp  25 sp

INCREASE YOUR TEAM'S VELOCITY AND GET MORE DONE WITH THE #1 SCRUM TOOL.

You can do better than sticky notes. **Axosoft Scrum** is the easiest way to manage backlogs, plan releases and analyze burndown charts. Learn more at [Axosoft.com/MSDNscrum](https://axosoft.com/MSDNscrum).



\$1



A BUG TRACKER FOR YOUR WHOLE TEAM NOW COSTS THE SAME AS A DOLLAR MENU BURGER.

Axosoft Bug Tracker is now just \$1 per year for your entire team. Not \$1 per user, but \$1 for everyone. Check it out at Axosoft.com/MSDNbugs.





dtSearch®

Instantly Search Terabytes of Text

25+ fielded and full-text search types

dtSearch's **own document filters** support "Office," PDF, HTML, XML, ZIP, emails (with nested attachments), and many other file types

Supports databases as well as static and dynamic websites

Highlights hits in all of the above

APIs for .NET, Java, C++, SQL, etc.

64-bit and 32-bit; Win and Linux

"lightning fast" Redmond Magazine

"covers all data sources" eWeek

"results in less than a second" InfoWorld

hundreds more reviews and developer case studies at www.dtsearch.com

dtSearch products:

Desktop with Spider	Web with Spider
Network with Spider	Engine for Win & .NET
Publish (portable media)	Engine for Linux
Document filters also available for separate licensing	

Ask about fully-functional evaluations

The Smart Choice for Text Retrieval® since 1991
www.dtSearch.com 1-800-IT-FINDS

msdn

magazine

JUNE 2014 VOLUME 29 NUMBER 6

MOHAMMAD AL-SABT Editorial Director/mmeditor@microsoft.com

KENT SHARKEY Site Manager

MICHAEL DESMOND Editor in Chief/mmeditor@microsoft.com

LAKE LOW Features Editor

SHARON TERDEMAN Features Editor

DAVID RAMEL Technical Editor

WENDY HERNANDEZ Group Managing Editor

SCOTT SHULTZ Creative Director

JOSHUA GOULD Art Director

SENIOR CONTRIBUTING EDITOR Dr. James McCaffrey

CONTRIBUTING EDITORS Rachel Appel, Dino Esposito, Kenny Kerr, Julie Lerman, Ted Neward, Charles Petzold, David S. Platt, Bruno Terkaly, Ricardo Villalobos

Redmond Media Group

Henry Allain President, Redmond Media Group

Michele Imgrund Sr. Director of Marketing & Audience Engagement

Tracy Cook Director of Online Marketing

Irene Fincher Audience Development Manager

ADVERTISING SALES: 818-674-3416/dlbianca@1105media.com

Dan LaBianca Vice President, Group Publisher

Chris Kourtoglou Regional Sales Manager

Danna Vedder Regional Sales Manager/Microsoft Account Manager

David Seymour Director, Print & Online Production

Anna Lyn Bayaia Production Coordinator/msdnadproduction@1105media.com

1105 MEDIA

Neal Vitale President & Chief Executive Officer

Richard Vitale Senior Vice President & Chief Financial Officer

Michael J. Valenti Executive Vice President

Christopher M. Coates Vice President, Finance & Administration

Erik A. Lindgren Vice President, Information Technology & Application Development

David F. Myers Vice President, Event Operations

Jeffrey S. Klein Chairman of the Board

MSDN Magazine (ISSN 1528-4859) is published monthly by 1105 Media, Inc., 9201 Oakdale Avenue, Ste. 101, Chatsworth, CA 91311. Periodicals postage paid at Chatsworth, CA 91311-9998, and at additional mailing offices. Annual subscription rates payable in US funds are: U.S. \$35.00, International \$60.00. Annual digital subscription rates payable in U.S. funds are: U.S. \$25.00, International \$25.00. Single copies/back issues: U.S. \$10, all others \$12. Send orders with payment to: *MSDN Magazine*, PO. Box 3167, Carol Stream, IL 60132, email MSDNmag@1105service.com or call (847) 763-9560. **POSTMASTER:** Send address changes to *MSDN Magazine*, PO. Box 2166, Skokie, IL 60076. Canada Publications Mail Agreement No: 40612608. Return Undeliverable Canadian Addresses to Circulation Dept. or XPO Returns: PO. Box 201, Richmond Hill, ON L4B 4R5, Canada.

Printed in the U.S.A. Reproductions in whole or part prohibited except by written permission. Mail requests to "Permissions Editor," c/o *MSDN Magazine*, 4 Venture, Suite 150, Irvine, CA 92618.

Legal Disclaimer: The information in this magazine has not undergone any formal testing by 1105 Media, Inc. and is distributed without any warranty expressed or implied. Implementation or use of any information contained herein is the reader's sole responsibility. While the information has been reviewed for accuracy, there is no guarantee that the same or similar results may be achieved in all environments. Technical inaccuracies may result from printing errors and/or new developments in the industry.

Corporate Address: 1105 Media, Inc., 9201 Oakdale Ave., Ste 101, Chatsworth, CA 91311, www.1105media.com

Media Kits: Direct your Media Kit requests to Matt Morollo, VP Publishing, 508-532-1418 (phone), 508-875-6622 (fax), mmorollo@1105media.com

Reprints: For single article reprints (in minimum quantities of 250-500), e-prints, plaques and posters contact: PARS International, Phone: 212-221-9595, E-mail: 1105reprints@parsintl.com, www.magreprints.com/QuickQuote.asp

List Rental: This publication's subscriber list, as well as other lists from 1105 Media, Inc., is available for rental. For more information, please contact our list manager, Jane Long, Merit Direct. Phone: 913-685-1301; E-mail: jljong@meritdirect.com; Web: www.meritdirect.com/1105

All customer service inquiries should be sent to MSDNmag@1105service.com or call 847-763-9560.



Printed in the USA



Use LEADTOOLS to develop world-class Document, Medical and Multimedia Imaging applications with support for **OCR, Forms Recognition, Barcode, PDF, DICOM, PACS, Viewers** and *more*.

.NET Windows API WinRT Linux iOS OS X Android HTML5/JavaScript

DOWNLOAD OUR 60 DAY EVALUATION
WWW.LEADTOOLS.COM



SALES@LEADTOOLS.COM
800.637.1840





Bleeding Heart

The Heartbleed vulnerability in the OpenSSL implementation has been decried as perhaps the greatest code security flaw the Internet has ever seen. The flaw potentially made secure connections created using OpenSSL an open book to third parties. And like so many software calamities, it was the result of a simple gaffe.

The Heartbeat Extension feature of OpenSSL was implemented in 2011 with a parameter that enabled clients to specify the size of the packet to be sent by the server in response to a Heartbeat Request message. But the feature failed to account for the actual size of the defined payload. A client could request a larger request message than the payload required. And because of the way OpenSSL allocated memory, the returned message could include the payload, plus whatever contents were currently allocated in the memory buffer at the time.

This is a Homer Simpson-level gaffe, and for two years, nobody noticed the gaping hole left in supposedly secure OpenSSL connections.

The result was a kind of reverse buffer overrun, where memory that wasn't supposed to be there got transmitted—unencrypted—back to the client. And, often, that excess memory space contained the private keys for connected Web sites.

This is a Homer Simpson-level gaffe, and for two years nobody noticed the gaping hole left in supposedly secure OpenSSL connections. Alex Papadimoulis, founder of consultancy Inedo and creator of the popular software blog The Daily WTF (thedailywtf.com), has for years chronicled the sad, frustrating and sometimes hilarious tales of software development gone wrong. He compared

Heartbleed to a national chain of banks forgetting to lock the vaults at night ... for a month.

"It takes a whole new level of bad to surprise me anymore, but that's exactly what Heartbleed delivered," Papadimoulis told me in an e-mail interview. "It wasn't your typical WTF, in that it was bad code created by a clueless coder. It was more the perfect illustration of compounding consequences from a simple bug."

The Heartbleed flaw serves as an urgent and humbling reminder that the history of software development is riddled with bonehead mistakes. I'm tempted to set a scale for astonishingly avoidable software flaws, with the top-most range set at Mars Climate Orbiter (MCO). MCO was a half-billion dollar-or-so mission to put an advanced satellite in orbit around Mars. It failed.

The \$328 million craft burned up in the Martian atmosphere because the NASA spacecraft team in Colorado and the mission navigation team in California used different units of measure (one English, the other metric) to calculate thrust and force. Space exploration may not be a game of inches, but it is a game of newton-seconds, and an astonishingly simple mismatch doomed the mission to failure.

Where on the MCO scale might Heartbleed fall? Somewhere around 0.85 MCOs, I think. The MCO failure cost half a billion dollars and denied us a decade of priceless, scientific exploration, but the actual costs of Heartbleed may never be fully disclosed or realized. While quick action likely prevented catastrophic damage, we do know that the Canada Revenue Agency reported the theft of critical data for some 900 taxpayers. And I have little doubt that additional disclosures will be in the offing.

"The scale of this vulnerability is what makes it so remarkable, but this type of bug is unavoidable," Papadimoulis says. "The best thing we can do is make our applications and systems easy to maintain and deploy, so that when bugs like this are discovered they can be patched without issue."

It's good advice. As long as humans create software, software will have flaws. So best to be ready to address those flaws when they emerge. Where do you think Heartbleed sits in the annals of botched software development? Email me at mmeditor@microsoft.com.

Visit us at msdn.microsoft.com/magazine. Questions, comments or suggestions for *MSDN Magazine*? Send them to the editor: mmeditor@microsoft.com.

© 2014 Microsoft Corporation. All rights reserved.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, you are not permitted to reproduce, store, or introduce into a retrieval system *MSDN Magazine* or any part of *MSDN Magazine*. If you have purchased or have otherwise properly acquired a copy of *MSDN Magazine* in paper format, you are permitted to physically transfer this paper copy in unmodified form. Otherwise, you are not permitted to transmit copies of *MSDN Magazine* (or any part of *MSDN Magazine*) in any form or by any means without the express written permission of Microsoft Corporation.

A listing of Microsoft Corporation trademarks can be found at microsoft.com/library/toolbar/3.0/trademarks/en-us.mspx. Other trademarks or trade names mentioned herein are the property of their respective owners.

MSDN Magazine is published by 1105 Media, Inc. 1105 Media, Inc. is an independent company not affiliated with Microsoft Corporation. Microsoft Corporation is solely responsible for the editorial contents of this magazine. The recommendations and technical guidelines in *MSDN Magazine* are based on specific environments and configurations. These recommendations or guidelines may not apply to dissimilar configurations. Microsoft Corporation does not make any representation or warranty, express or implied, with respect to any code or other information herein and disclaims any liability whatsoever for any use of such code or other information. *MSDN Magazine*, *MSDN*, and Microsoft logos are used by 1105 Media, Inc. under license from owner.

Data Quality Tools for Developers

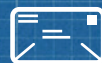
A better way to build in data verification

Since 1985, Melissa Data has provided the tools developers need to enhance databases with clean, correct, and current contact data. Our powerful, yet affordable APIs and Cloud services provide maximum flexibility and ease of integration across industry-standard technologies, including .NET, Java, C, and C++. Build in a solid framework for data quality and protect your investments in data warehousing, business intelligence, and CRM.

- Verify international addresses for over 240 countries
- Enhance contact data with phone numbers and geocodes
- Find, match, and eliminate duplicate records
- Sample source code for rapid application development
- Free trials with 120-day ROI guarantee

Melissa Data.

Architecting data quality success.



Address
Verification



Phone
Verification



Email
Verification



Geocoding



Matching/
Dedupe



Change of
Address

MELISSA DATA®

www.MelissaData.com 1-800-MELISSA



External Authentication with ASP.NET Identity

Visual Studio 2013 includes new ASP.NET MVC 5 and Web Forms project templates for building applications. The sample templates might optionally contain a default authentication layer based on ASP.NET Identity. The code generated for you contains a lot of features, but it might not be that easy to read. In this article, I'll show how to perform authentication through an external OAuth- or OpenID-based server such as a social network and the steps you might want to take once authentication is complete and you have access to all the downloaded claims.

I'll start with an empty ASP.NET MVC 5 application and add the bare minimum authentication layer an app might need. You'll see the new ASP.NET Identity framework brings a new level of complexity and power to the whole authentication stack, but, at the same time, it makes it quick and easy to arrange for external user authentication without any need for you to learn new packages and APIs.

Factoring Out the Account Controller

Since the very first release of ASP.NET MVC, the sample application that represents the face of the framework to new users offered a rather bloated controller class to manage all aspects of authentication. In my March 2014 column, "A First Look at ASP.NET Identity" (msdn.microsoft.com/magazine/dn605872), I provided an overview of the ASP.NET Identity framework starting from the code you get by default in Visual Studio. This time, instead, I'll put the various pieces together starting from an empty ASP.NET MVC 5 project. In doing so, I'll carefully follow the Single Responsibility Principle (bit.ly/1gGjFtx).

I'll start with an empty project and then add some ASP.NET MVC 5 scaffolding. Next, I'll add two new controller classes: LoginController and AccountController. The former is only concerned with sign-in and sign-out operations whether accomplished through the site-specific, local membership system or an external OAuth provider such as Twitter. As the name itself suggests, the AccountController class includes any other functionality mostly related to the management of the user accounts. For the sake of simplicity, I'll only include the ability to register a new user at this time. **Figure 1** shows the skeleton of the two controller classes, as they appear in Visual Studio 2013.

The UI is articulated in three partial views: logged user, local login form and social login form, as shown in **Figure 2**.

The IdentityController Class

In the skeleton shown in **Figure 1**, both the LoginController and AccountController classes inherit from the base ASP.NET MVC 5 Controller class. If you're going to use the ASP.NET Identity framework, however, this might not be the ideal choice, as it could lead to some duplicated code. I suggest you create a temporary project in Visual Studio 2013 and take a look at the default code stored in the almighty and all-encompassing AccountController class. ASP.NET Identity requires you to inject the `userManager<TUser>` class and the storage mechanism into the controller. In addition, you might want to have a couple of helper methods to simplify the process of signing users in and redirecting. You also need a couple of properties to reference the UserManager root object and maybe

yet another helper property to establish a point of contact with the underlying Open Web Interface for .NET (OWIN) middleware.

It's probably a good idea to introduce an intermediate class where all this boilerplate code can be comfortably stuffed once and only once. I'll call this class IdentityController and define it as shown in **Figure 3**.

The use of generics—and related constraints—binds the IdentityController class to use any definition of a user based on the IdentityUser class and any definition of a storage context based on IdentityDbContext.

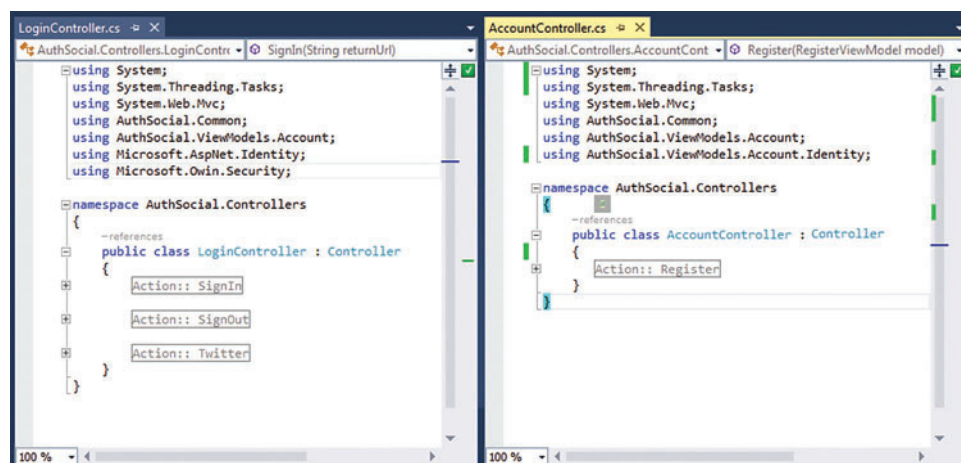


Figure 1 Skeleton of LoginController and AccountController Classes in Visual Studio 2013

IdentityUser, in particular, is just the basic class to describe a user and offer a minimal set of properties. You're welcome to create new and more specific classes through plain inheritance. When you do this, you just specify your custom class in the declaration of classes derived from IdentityController. Here's the declaration of LoginController and AccountController based on IdentityController<TUser, TDbContext>:

```
public class LoginController :
    IdentityController<IdentityUser, IdentityDbContext>
{
    ...
}
public class AccountController :
    IdentityController<IdentityUser, IdentityDbContext>
{
    ...
}
```

In this way, you set the groundwork for partitioning most of the complexity of authentication and user management code across three classes. Moreover, you now have one class to provide common services and one class that just focuses on login tasks. Finally, you have a separate class to deal with management of user accounts. I'll just start from the account class, where, for the sake of simplicity, I only expose an endpoint to register new users.

The AccountController Class

Generally, you need a controller method only if there's some piece of displayed UI with which the end user interacts. If there's, say, a submit button the user can click, then you need a POST-sensitive controller method. In order to define this method, start from the view model for it. The view model class is a plain Data Transfer Object (DTO) that gathers together any data (strings, dates, numbers, Booleans and collections) that goes in and out of the displayed UI. For a method that adds a new user into the local membership system, you can have the following view model class (the source code is the same as what you get from the default Visual Studio 2013 ASP.NET MVC 5 template):

```
public class RegisterViewModel
{
    public string UserName { get; set; }
    public string Password { get; set; }
    public string ConfirmPassword { get; set; }
}
```

You might also want to decorate this class with data annotations for validation and display purposes. At a higher design level, though, all that matters is that the class features one property for each input control in the actual UI. **Figure 4** presents the implementation of the controller method that handles the POST request to register a new user.

The CreateAsync method on the UserManager class just uses the underlying storage mechanism to create a new entry for the specified user. This AccountController class is also the ideal place to define other methods to edit or just delete user accounts and change or reset passwords.

The LoginController Class

The login controller will feature methods to sign users in and out in any way that's acceptable for the application. Similarly to what

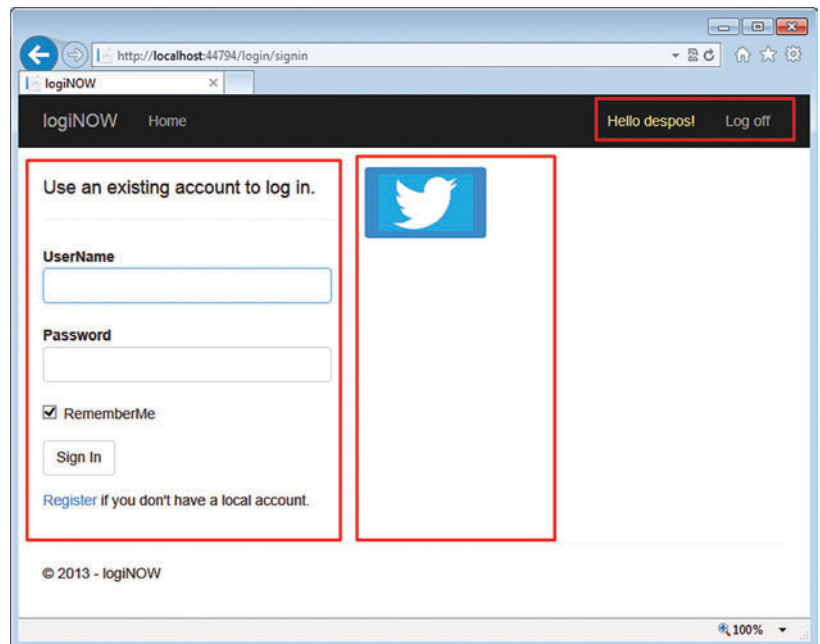


Figure 2 The Three Partial Views Used in the Authentication Stack

I discussed about the account controller, the SignIn method will manage a DTO with properties to carry credentials, a flag for the authentication persistence and perhaps a return URL. In the default code, the return URL is managed through the ViewBag; you can stuff it in the view model class as well:

```
public class LoginViewModel
{
    public string UserName { get; set; }
    public string Password { get; set; }
    public bool RememberMe { get; set; }
    public string returnUrl { get; set; }
}
```

The code to interact with the local membership system, entirely managed on the application server, is pasted in from the default project. Here's an excerpt from the POST implementation of the new SignIn method:

```
var user = await UserManager.FindAsync(
    model.UserName, model.Password);
if (user != null)
{
    await SignInAsync(user, model.RememberMe);
    return RedirectToLocal(returnUrl);
}
```

The code that ultimately signs in users is a method—named SignInAsync—borrowed from the default implementation and rewritten as a protected method in the IdentityController base class:

```
protected async Task SignInAsync(TUser user, bool isPersistent)
{
    AuthenticationManager.SignOut(
        DefaultAuthenticationTypes.ExternalCookie);
    var identity = await UserManager.CreateIdentityAsync(user,
        DefaultAuthenticationTypes.ApplicationCookie);
    AuthenticationManager.SignIn(
        new AuthenticationProperties { IsPersistent = isPersistent },
        identity);
}
```

When using the ASP.NET Identity framework, you don't use any class that's explicitly related to the ASP.NET machinery. In the context of an ASP.NET MVC host, you still manage user authentication through login forms and ASP.NET authentication cookies—except that you don't use the FormsAuthentication ASP.NET class directly.

There are two layers of code with which classic ASP.NET developers need to cope. One is the plain ASP.NET Identity API represented by classes such as `userManager` and `identityuser`. The other is the OWIN cookie middleware, which just shields the façade authentication API from the nitty-gritty details of how a logged user's information is stored and persisted. Forms authentication is still used, and the old acquaintance, the `.ASPXAUTH` cookie, is still created. However, everything now happens behind the curtain of a bunch of new methods. For a deeper understanding of this, see Brock Allen's blog post, "A Primer on OWIN Cookie Authentication Middleware for the ASP.NET Developer," at bit.ly/1fKG0G9.

Figure 3 A New Base Class for ASP.NET Identity Authentication

```
using System.Collections.Generic;
using System.Security.Claims;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;
using AuthSocial.ViewModels.Account.Identity;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.Owin.Security;

namespace AuthSocial.Controllers
{
    public class IdentityController<TUser, TDbContext> : Controller
        where TUser : IdentityUser
        where TDbContext : IdentityDbContext, new()
    {
        public IdentityController()
            : this(new UserManager<TUser>(
                new UserStore<TUser>(new TDbContext())))
        {
        }
        public IdentityController(UserManager<TUser> userManager)
        {
            UserManager = userManager;
        }

        protected UserManager<TUser> UserManager { get; set; }

        protected IAuthenticationManager AuthenticationManager
        {
            get { return HttpContext.GetOwinContext().Authentication; }
        }

        protected async Task SignInAsync(
            TUser user, bool isPersistent)
        {
            AuthenticationManager.SignOut(
                DefaultAuthenticationTypes.ExternalCookie);
            var identity = await UserManager.CreateIdentityAsync(user,
                DefaultAuthenticationTypes.ApplicationCookie);
            AuthenticationManager.SignIn(
                new AuthenticationProperties {
                    IsPersistent = isPersistent },
                identity);
        }

        protected ActionResult RedirectToLocal(string returnUrl)
        {
            if (Url.IsLocalUrl(returnUrl))
            {
                return Redirect(returnUrl);
            }
            return RedirectToAction("Index", "Home");
        }

        protected ClaimsIdentity GetBasicUserIdentity(string name)
        {
            var claims = new List<Claim> {
                new Claim(ClaimTypes.Name, name) };
            return new ClaimsIdentity(
                claims, DefaultAuthenticationTypes.ApplicationCookie);
        }
    }
}
```

Look at the following code you find invoked right from `Startup.Auth.cs`, the entry point for OWIN middleware:

```
app.UseCookieAuthentication(
    new CookieAuthenticationOptions
    {
        AuthenticationType =
            DefaultAuthenticationTypes.ApplicationCookie,
        LoginPath = new PathString("/Login/SignIn")
    });

app.UseExternalSignInCookie(
    DefaultAuthenticationTypes.ExternalCookie);
app.UseTwitterAuthentication(
    consumerKey: "45c6...iQ",
    consumerSecret: "pRcoXT...kdnU");
```

This code is a sort of a fluent interface you use instead of the old-fashioned `web.config` authentication entries. In addition, it also links in Twitter-based authentication and configures the OWIN middleware to use the `ExternalSignInCookie` in order to temporarily store information about a user logging in through a third-party login provider.

The object `AuthenticationManager` represents the entry point in the OWIN middleware and the façade through which you invoke the underlying ASP.NET authentication API based on `FormsAuthentication`.

The net effect is you end up using a unified API to authenticate users and verify credentials whether the membership system is local to your Web server or externally delegated to a social network

Figure 4 Registering a New User

```
public async Task<ActionResult> Register(
    RegisterViewModel model)
{
    if (ModelState.IsValid) {
        var user = new IdentityUser { UserName = model.UserName };
        var result = await UserManager.CreateAsync(
            user, model.Password);
        if (result.Succeeded) {
            await SignInAsync(user, false);
            return RedirectToAction("Index", "Home");
        }
        Helpers.AddErrors(ModelState, result);
    }
    return View(model);
}
```

Figure 5 The ChallengeResult Class

```
public class ChallengeResult : HttpUnauthorizedResult
{
    public ChallengeResult(string provider, string redirectUri)
    {
        LoginProvider = provider;
        RedirectUri = redirectUri;
    }

    public string LoginProvider { get; set; }
    public string RedirectUri { get; set; }

    public override void ExecuteResult(ControllerContext context)
    {
        var properties = new AuthenticationProperties
        {
            RedirectUri = RedirectUri
        };
        context.HttpContext
            .GetOwinContext()
            .Authentication
            .Challenge(properties, LoginProvider);
    }
}
```


Empower Your Customers



Create & Edit PDFs in .Net - ActiveX - WinRT

- Edit, process and print PDF 1.7 documents
- Create, fill-out and annotate PDF forms
- Fast and lightweight 32- and 64-bit components for .Net and ActiveX/COM
- New WinRT Component enables publishing C#, C++CX or Javascript apps to Windows Store
- New Postscript/EPS to PDF conversion module



Complete Suite of Accurate PDF Components

- All your PDF processing, conversion and editing in a single package
- Combines Amyuni PDF Converter and PDF Creator for easy licensing, integration and deployment
- Includes our Microsoft certified PDF Converter printer driver
- Export PDF documents into other formats such as JPeg, PNG, XAML or HTML5



Advanced HTML to PDF & XAML

- Direct conversion of HTML files into PDF and XAML without the use of a web browser or a printer driver
- Easy Integration and deployment within developer's applications
- WebkitPDF is based on the Webkit Open Source library and Amyuni PDF Creator



High Performance PDF Printer For Desktops and Servers



- Our high-performance printer driver optimized for Web, Application and Print Servers. Print to PDF in a fraction of the time needed with other tools. WHQL tested for Windows 32 and 64-bit including Windows Server 2012 and Windows 8
- Standard PDF features included with a number of unique features. Interface with any .Net or ActiveX programming language
- Easy licensing and deployment to fit system administrator's requirements



AMYUNI

All development tools available at

www.amyuni.com

USA and Canada

Toll Free: 1866 926 9864
Support: 514 868 9227
sales@amyuni.com

Europe

UK: 0800-015-4682
Germany: 0800-183-0923
France: 0800-911-248

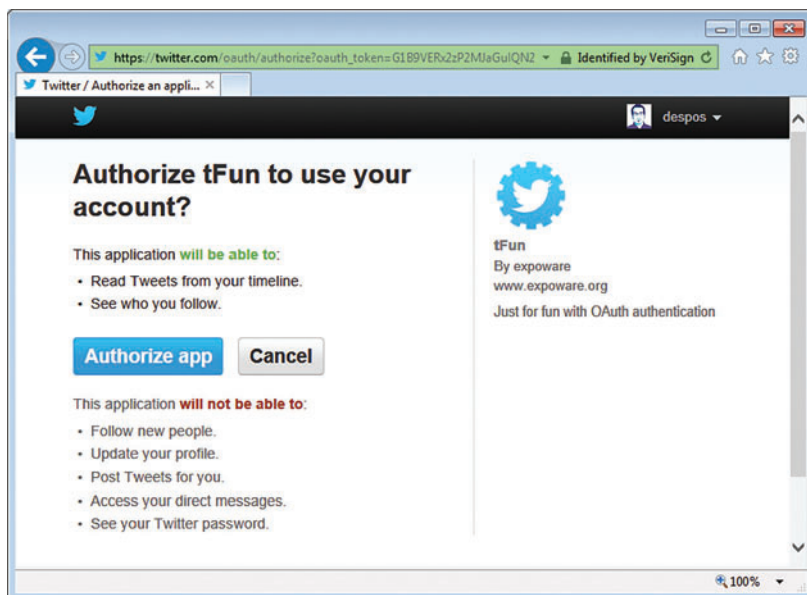


Figure 6 The Classic Twitter-Based Authentication

login provider. Here's the code you use if it's acceptable that users of your site authenticate via their Twitter accounts:

```
[AllowAnonymous]
public ActionResult Twitter(String returnUrl)
{
    var provider = "Twitter";
    return new ChallengeResult(provider,
        Url.Action("ExternalLoginCallback", "Login",
            new { ReturnUrl = returnUrl }));
}
```

First and foremost, you need to have a Twitter application configured and associated with a Twitter developer account. For more information, see dev.twitter.com. A Twitter application is uniquely identified by a pair of alphanumeric strings known as consumer key and secret.

The default code you get from the ASP.NET MVC 5 project wizard also suggests you use an entirely new class—`ChallengeResult`—to handle external logins, shown in Figure 5.

The class delegates to the OWIN middleware—in the method named `Challenge`—the task to connect to the specified login provider (registered at startup) to carry out authentication. The return URL is the method `ExternalLoginCallback` on the login controller:

```
[AllowAnonymous]
public async Task<ActionResult> ExternalLoginCallback(
    string returnUrl)
{
    var info = await AuthenticationManager.GetExternalLoginInfoAsync();
    if (info == null)
        return RedirectToAction("SignIn");

    var identity = GetBasicUserIdentity(info.DefaultUserName);
    AuthenticationManager.SignIn(
        new AuthenticationProperties { IsPersistent = true },
        identity);
    return RedirectToLocal(returnUrl);
}
```

The method `GetExternalLoginInfoAsync` retrieves information about the identified user that the login provider makes available. When the method returns, the application has enough information to actually sign the user in, but nothing has happened yet. It's important to be aware of the options you have here. First, you can

simply proceed and create a canonical ASP.NET authentication cookie, as in my example. Defined in the `IdentityController` base class, the helper method `GetBasicUserIdentity` creates a `ClaimsIdentity` object around the provided Twitter account name:

```
protected ClaimsIdentity GetBasicUserIdentity(string name)
{
    var claims = new List<Claim> { new Claim(
        ClaimTypes.Name, name) };
    return new ClaimsIdentity(
        claims, DefaultAuthenticationTypes.ApplicationCookie);
}
```

Once the Twitter user authorizes the Twitter application to share information, the Web page receives the user name and potentially more data, as shown in Figure 6 (this depends on the actual social API being used).

Notice in the preceding code that in the `ExternalLoginCallback` method no user is added to the local membership system. This is the simplest scenario. In other situations, you might want to use the Twitter information to programmatically register a link between a local login and the

externally authenticated user name (this is the solution presented in the default authentication code from the wizard). Finally, you can decide to redirect users to a different page to let them enter an e-mail address or just register for your site.

ASP.NET Identity makes it
easy to code and mix classic and
social authentication behind
a unified façade.

Mixing Classic and Social Authentication

Some level of authentication is necessary in nearly every Web site. ASP.NET Identity makes it easy to code and mix classic and social authentication behind a unified façade. It's important, though, that you have a clear strategy about the way you want to handle accounts for users and what information you want to collect and store about them. Mixing classic and social authentication means you can have an individual user account and associate it with multiple login providers such as Facebook, Twitter and all other login providers supported by ASP.NET Identity. In this way, users have multiple ways to sign in, but they remain unique to the eyes of your application. And more login providers can be added or removed as is convenient. ■

DINO ESPOSITO is the author of *Architecting Mobile Solutions for the Enterprise* (Microsoft Press, 2012) and the upcoming *Programming ASP.NET MVC 5* (Microsoft Press). A technical evangelist for the .NET Framework and Android platforms at JetBrains and frequent speaker at industry events worldwide, Esposito shares his vision of software at software2cents.wordpress.com and on Twitter at twitter.com/despos.

THANKS to the following Microsoft technical expert for reviewing this article:
Pranav Rastogi

NEW HOSTING

POPULAR APPS - NOW EVEN BETTER!



WordPress & Powerful App Platforms!

- Over 140 popular apps including WordPress, Drupal™, Joomla!™, TYPO3 and more
- 2 GB of RAM guaranteed – ideal for app performance
- **New:** Security & version upgrade notifications
- App Expert Support

Powerful Tools

- **PHP 5.5,** Perl, Python, Ruby
- 1&1 Mobile Website Builder
- NetObjects Fusion® 2013 included

Successful Marketing

- Facebook® advertising credits
- Listing in business directories
- 1&1 Search Engine Optimization
- 1&1 E-Mail Marketing Manager

State-of-the-Art Technology

- **Maximum Availability (Geo-Redundancy)**
- 300 Gbit/s network connection
- Maximum performance with 1&1 CDN powered by CloudFlare™

All Inclusive

- **1 FREE** domain: .com, .net, .org, .biz, .info
- Unlimited power: webspace, traffic, mail accounts, MySQL databases
- Secure e-mail addresses with virus and spam protection
- Linux or Windows operating system

**COMPLETE
PACKAGES
FOR PROFESSIONALS**

Starting at

\$0.99
per year*



1 (877) 461-2631



1and1.com

*Offer valid for a limited time only. The \$0.99/year price reflects a 12-month pre-payment option for the 1&1 Basic Linux Hosting Package. Regular price of \$5.99/month after 12 months. Some features listed are only available with package upgrade. Visit www.1and1.com for full promotion details. Program and pricing specifications and availability subject to change without notice. 1&1 and the 1&1 logo are trademarks of 1&1 Internet, all other trademarks are the property of their respective owners. ©2014 1&1 Internet. All rights reserved.

Working with Files?

Convert Print Create Combine Modify



Aspose.Words

DOC, DOCX, RTF, HTML, PDF, XPS & other document formats.



Aspose.Pdf

PDF, XML, XLS-FO, HTML, BMP, JPG, PNG & other image formats.



Aspose.Cells

XLS, XLSX, XLSM, XLTX, CSV, SpreadsheetML & image formats.



Aspose.Slides

PPT, PPTX, POT, POTX, XPS, HTML, PNG, PDF & other formats.



Aspose.Email

MSG, EML, PST, EMLX & other formats.



Aspose.BarCode

JPG, PNG, BMP, GIF, TIFF, WMF, ICON & other image formats.



Aspose.Imaging

PDF, BMP, JPG, GIF, TIFF, PNG, PSD & other image formats.



Aspose.Tasks

XML, MPP, SVG, PDF, TIFF, PNG, CSV, MPT & other formats.



Aspose.Diagram

VSD, VSDX, VSS, VST, VSX & other formats.



Aspose.Note

ONE, PNG, JPG, BMP, GIF & PDF

... and more!

100% Standalone - No Office Automation



NET Libraries



Java Libraries



Cloud APIs



Android Libraries



Scan for a 20% saving!



Collaborating with Files?

View Sign Annotate Assemble Compare



GroupDocs.Viewer

Native-text, high-fidelity HTML5 document viewer with support for over 45 file formats.



GroupDocs.Signature

Electronic signature API that gives your apps legally binding e-signature capabilities.



GroupDocs.Conversion

Universal document converter for fast conversion between more than 45 file formats.



GroupDocs.Annotation

A powerful API that lets developers annotate Microsoft Office, PDF and other documents within their own apps.



GroupDocs.Assembly

Incorporates data entered by users through online forms Into both Microsoft Office and PDF documents.



GroupDocs.Comparison

A diff view API that allows end users to quickly find differences between two revisions of a document.

100% Standalone - No Office Automation



NET Libraries



Java Libraries



Cloud APIs



Cloud Apps

document collaboration APIs



GROUPDOCS
Your Document Collaboration APIs

SALES INQUIRIES: +1 214 329 9760 sales@groupdocs.com www.groupdocs.com



High-Performance Window Layering Using the Windows Composition Engine

I've been fascinated by layered windows since I first came across them in Windows XP. The ability to escape the rectangular, or near-rectangular, bounds of a traditional desktop window always seemed so intriguing to me. And then Windows Vista happened. That much-maligned release of Windows hinted at the start of something far more exciting and far more flexible. Windows Vista started something we've only begun to appreciate now that Windows 8 is here, but it also marked the slow decline of the layered window.

Windows Vista introduced a service called the Desktop Window Manager. The name was and continues to be misleading. Think of it as the Windows composition engine or compositor. This composition engine completely changed the way application windows are rendered on the desktop. Rather than allowing each window to render directly to the display, or display adapter, every window renders to an off-screen surface or buffer. The system allocates one such surface per top-level window and all GDI, Direct3D and, of course, Direct2D graphics are rendered to these surfaces. These off-screen surfaces are called redirection surfaces because GDI drawing commands and even Direct3D swap chain presentation requests are redirected or copied (within the GPU) to the redirection surface.

At some point, independent of any given window, the composition engine decides it's time to compose the desktop given the latest batch of changes. This involves composing all of these redirection surfaces together, adding the non-client areas (often called window chrome), perhaps adding some shadows and other effects, and presenting the final result to the display adapter.

This composition process has many wonderful benefits I'll explain over the next few months as I explore Windows composition in more detail, but it also has one potentially serious restriction in that these redirection surfaces are opaque. Most applications are quite happy with this and it certainly makes a lot of sense from a performance perspective, because alpha blending is expensive. But this leaves layered windows out in the cold.

If I want a layered window, I have to take a performance hit. I describe the specific architectural limitations in my column, "Layered Windows with Direct2D" (msdn.microsoft.com/magazine/ee819134). To summarize, layered windows are processed by the CPU, primarily to support hit testing of alpha-blended pixels. This means the CPU needs a copy of the pixels that make up the layered window's surface area. Either I render on the CPU, which tends to be a lot slower than GPU rendering, or I render on the GPU, in which case I must pay the bus bandwidth tax because everything I render must be copied from video memory to system

memory. In the aforementioned column, I also show how I might make the most of Direct2D to squeeze as much performance as possible out of the system because only Direct2D lets me make the choice between CPU and GPU rendering. The kicker is that even though the layered window necessarily resides in system memory, the composition engine immediately copies it to video memory such that the actual composition of the layered window is still hardware-accelerated.

While I can't offer you any hope of traditional layered windows returning to prominence, I do have some good news. Traditional layered windows offer two specific features of interest. The first is

Figure 1 Creating a Traditional Window

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>

#pragma comment(lib, "user32.lib")

int __stdcall wWinMain(HINSTANCE module, HINSTANCE, PWSTR, int)
{
    WNDCLASS wc = {};

    wc.hCursor    = LoadCursor(nullptr, IDC_ARROW);
    wc.hInstance  = module;
    wc.lpszClassName = L"window";
    wc.style      = CS_HREDRAW | CS_VREDRAW;

    wc.lpfnWndProc =
    [] (HWND window, UINT message, WPARAM wparam, LPARAM lparam) -> LRESULT
    {
        if (WM_DESTROY == message)
        {
            PostQuitMessage(0);
            return 0;
        }

        return DefWindowProc(window, message, wparam, lparam);
    };

    RegisterClass(&wc);

    HWND const window = CreateWindow(wc.lpszClassName, L"Sample",
                                     WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                                     CW_USEDEFAULT, CW_USEDEFAULT,
                                     CW_USEDEFAULT, CW_USEDEFAULT,
                                     nullptr, nullptr, module, nullptr);

    MSG message;

    while (BOOL result = GetMessage(&message, 0, 0, 0))
    {
        if (-1 != result) DispatchMessage(&message);
    }
}
```



Desktop. Web. Mobile. Your next great app starts here.

From interactive Desktop applications, to immersive Web and Mobile solutions, development tools built to meet your needs today and ensure your continued success tomorrow.

Download your free 30-day trial today and Experience the DevExpress Difference.

www.devexpress.com/try

DevExpress®

WIN ASP WPF SL VCL XAF CR

All trademarks or registered trademarks are property of their respective owners.

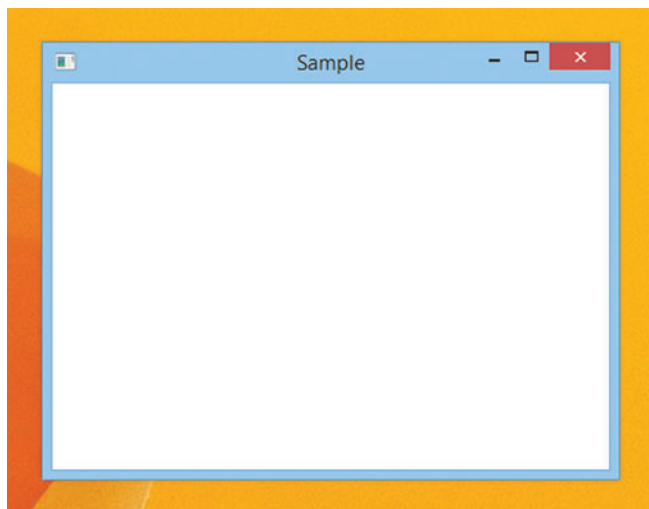


Figure 2 A Traditional Window on the Desktop

per-pixel alpha blending. Whatever I render to the layered window will be alpha blended with the desktop and with whatever is behind the window at any given moment. The second is the ability for User32 to hit test layered windows based on pixel alpha values, allowing mouse messages to fall through if the pixel at a particular point is transparent. As of Windows 8 and 8.1, User32 hasn't changed significantly, but what has changed is the ability to support per-pixel alpha blending purely on the GPU and without the cost of transmitting the window surface to system memory. This means I can now produce the effect of a layered window without compromising performance, provided I don't need per-pixel hit testing. The whole window will hit test uniformly. Setting aside hit testing, this excites me because it's something the system can obviously do, but it's just never been possible for applications to tap into this power. If this sounds intriguing to you, then read on and I'll show you how it's done.

The key to making this happen involves embracing the Windows composition engine. The composition engine first surfaced in Windows Vista as the Desktop Window Manager with its limited API and its popular translucent Aero glass effect. Then Windows 8 came along and introduced the DirectComposition API. This is just a more extensive API for the same composition engine. With the Windows 8 release, Microsoft finally allowed third-party developers to tap into the power of this composition engine that's been around for such a long time. And, of course, you'll need to embrace a Direct3D-powered graphics API such as Direct2D. But first you need to deal with that opaque redirection surface.

As I mentioned earlier, the system allocates one redirection surface for each top-level window. As of Windows 8, you can now create a top-level window and request that it be created without a redirection surface. Strictly speaking, this has nothing to do with layered windows, so don't use the `WS_EX_LAYERED` extended window style. (Support for layered windows actually gained a minor improvement in Windows 8, but I'll take a closer look at that in an upcoming column.) Instead, you need to use the `WS_EX_NOREDIRECTIONBITMAP` extended window style that tells the composition engine not to allocate a redirection surface for the window. I'll start with a simple and traditional desktop window. **Figure 1** provides an

example of filling in a `WNDCLASS` structure, registering the window class, creating the window and pumping window messages. Nothing new here, but these fundamentals continue to be essential. The window variable goes unused, but you'll need that in a moment. You can copy this into a Visual C++ project inside Visual Studio, or just compile it from the command prompt as follows:

```
cl /W4 /nologo Sample.cpp
```

Figure 2 shows you what this looks like on my desktop. Notice there's nothing unusual here. While the example provides no painting and rendering commands, the window's client area is opaque and the composition engine adds the non-client area, the border and title bar. Applying the `WS_EX_NOREDIRECTIONBITMAP` extended window style to get rid of the opaque redirection surface that represents this client area is a simple matter of switching out the `CreateWindow` function for the `CreateWindowEx` function with its leading parameter that accepts extended window styles:

```
HWND const window = CreateWindowEx(WS_EX_NOREDIRECTIONBITMAP,
    wc.lpszClassName, L"Sample",
    WS_OVERLAPPEDWINDOW | WS_VISIBLE,
    CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, CW_USEDEFAULT,
    nullptr, nullptr, module, nullptr);
```

The only things that changed are the addition of the leading argument, the `WS_EX_NOREDIRECTIONBITMAP` extended window style and, of course, the use of the `CreateWindowEx` function instead of the simpler `CreateWindow` function. The results on the desktop are, however, far more radical. **Figure 3** shows what this looks like on my desktop. Notice the window's client area is now completely transparent. Moving the window around will illustrate this fact. I can even have a video playing in the background and it won't be obscured in the least. On the other hand, the entire window hit tests uniformly and the window focus isn't lost when clicking within the client area. That's because the subsystem responsible for hit testing and mouse input isn't aware the client area is transparent.

Of course, the next question is how can you possibly render anything to the window if there's no redirection surface to provide to the composition engine? The answer comes from the DirectComposition API and its deep integration with the DirectX Graphics Infrastructure (DXGI). It's the same technique

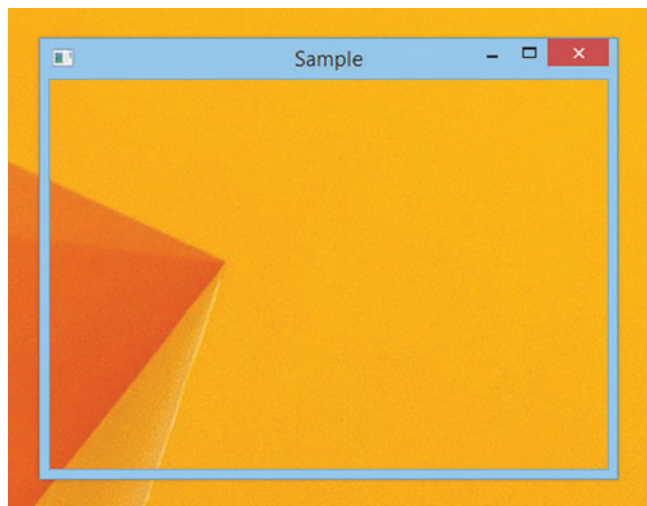


Figure 3 A Window Without a Redirection Surface



Become a UI Superhero

And deliver amazing user-experiences across
.NET platforms and next-gen mobile devices.



UI Controls for Desktop | Web | Mobile

Become a UI Superhero and deliver elegant solutions that fully address customer needs today and leverage your existing knowledge to build next generation touch-enabled solutions for tomorrow. Whether it's an Office-inspired application or a data centric analytics dashboard, DevExpress Universal ships with everything you'll need to build your best, without limits or compromise.



Office-Inspired Apps



Touch-Enabled Windows
& Web Apps



HTML5 Mobile Apps



Reporting-Dashboard
& Analytics Apps

When Only the Best Will Do

At DevExpress, we understand that you want the best and you don't have time to waste...that you are under tight budgets, tight deadlines and under pressure to deliver highly-functional apps that must run across devices and platforms. We know you are concerned about the future and want a comprehensive toolset that will never leave you behind. A toolset that will evolve over time so you can always leverage your existing code investments in order to meet any business challenge head-on.

We also know that you have many choices in the marketplace and we encourage you to evaluate our tools and compare them to the competition and decide for yourself. Download a free 30 day evaluation of DevExpress Universal from DevExpress.com/Try and see why your peers consistently vote DevExpress products #1.



Given the completeness and maturity of the many components in this year's packages, the Libraries category was tough going for the judges. However, one vendor's component suite made a stunning impact with its leaps in forward-thinking and polished presentation. DevExpress has been in the running for the top slot in the Jolt Libraries award for some time, but this year they have captured lightning in a bottle and were deemed the winner by a large margin.

—Mike Riley for Dr. Dobbs Magazine

UI Controls for

Your Next Great Desktop App



Deliver amazing user experiences and emulate the UI of today's most popular productivity apps with the 200+ controls that ship as part of the DevExpress WinForms and WPF Subscriptions. Whether you need to reproduce the look and feel of Microsoft Office or to deliver high-powered data mining and decision support systems for your enterprise, DevExpress libraries for the .NET Framework have been built so you can unleash the power of information and intuitively present it to your end-users in the shortest possible time.

And because your customers expect superior performance, regardless of dataset size or information complexity, DevExpress WinForms and WPF Controls are powered by a robust server-side data processing engine so that regardless of dataset size, your apps will always remain responsive and allow users to shape and analyze information at lightning speed.

“

Thank you DevExpress for building one of the most powerful, feature rich control suites on the market. Your superior support staff is second to none! I can't even begin to measure the amount of time the controls have saved me. And to see the look on people's faces when they experience the controls first hand, is priceless!

—Chris Todd, Software Developer

The screenshot displays a desktop application titled "Employees - DevExpress". The interface includes a top navigation bar with tabs like "HOME", "VIEW", and "ACTIONS". Below this is a search bar and a list of employees. A sidebar on the left shows a tree view of employee categories. A modal window titled "Loan Calculator" is overlaid on the right side of the application, showing a form with fields for purchase price, cash down, trade allowance, and amount owed, along with a summary of loan details and a total cost of \$6,376.60.

DEPARTMENT	FULL NAME	ADDRESS	CITY	STATE	ZIPCODE	EMAIL
Support (Count=7)	<input type="checkbox"/> Jackie Garmin	807 West Paseo Del Mar	Los Angeles	CA	90731	jackie@dv-email.com
	<input type="checkbox"/> Gabe Jones	1008 Elden Way	Beverly Hills	CA	90210	gabrie@dv-email.com
	<input type="checkbox"/> Nat Maguire	6400 E Bivby Hill Rd	Long Beach	CA	90815	natalie@dv-email.com
	<input type="checkbox"/> Antony Remmen	1542 S Beacon St	San Pedro	CA	90731	antonyr@dv-email.com
	<input type="checkbox"/> James Anderson	4800 Hollywood Blvd				
	<input type="checkbox"/> Kelly Rodriguez	1601 W Mountain				
	<input type="checkbox"/> Barb Banks	17955 Pacific Coast				
Shipping (Count=7)	<input type="checkbox"/> Dallas Lou	1 Bunker Hill				
	<input type="checkbox"/> Jenny Hobbs	205 Chautauqua B				
	<input type="checkbox"/> Robin Cosworth	501 N Main St.				
	<input type="checkbox"/> Mary Stern	113 N Cedar St.				
	<input type="checkbox"/> Victor Norris	811 West 7th St.				
	<input type="checkbox"/> Davey Jones	391 S Orange Grov				
	<input type="checkbox"/> Kevin Carter	424 N Main St.				
Sales (Count=10)	<input type="checkbox"/> Lincoln Bartlett	800 N Alameda St				
	<input type="checkbox"/> Todd Hoffman	2647 Arroyo Rd				
	<input type="checkbox"/> Clark Morgan	4202 Alhambra Ave				
	<input type="checkbox"/> Harv Mudd	351 Pacific St				
	<input type="checkbox"/> Hannah Brooklyn	536 Marsh Street				
	<input type="checkbox"/> Jim Packard	3801 Chester Ave				
	<input type="checkbox"/> Lucy Ball	203 Chautauqua B				
<input type="checkbox"/> Olivia Peyton	807 W Paseo Del M					

General Loan Information		Auto Loan Information	
Purchase price (before tax)	\$45,000.00	Interest rate	5.50%
Cash down	\$2,000.00	Auto Loan Details	
Trade allowance	\$15,000.00	Monthly payment	\$83.00
Amount owed on trade-in	\$13,000.00	Total Interest (5.50%)	\$6,376.60
Term in months	60	Potential Income tax savings	\$0.00
Sales tax rate	8.75%	Loan total	\$50,000.00
Fees	\$30.00	Cost of loan	\$6,376.60
Does your state deduct your trade-in value before calculating sales tax	Yes		
General Loan Details		Cost of Loan	
Purchase Amount	\$45,000.00	\$6,376.60	
Sales Tax	\$2,625.00		
Sales tax based on \$30,000.00 (sales tax deduction for trade-in allowance)			

Learn more and download your free 30-day trial
visit DevExpress.com/UISuperhero

Your Next Great Web App



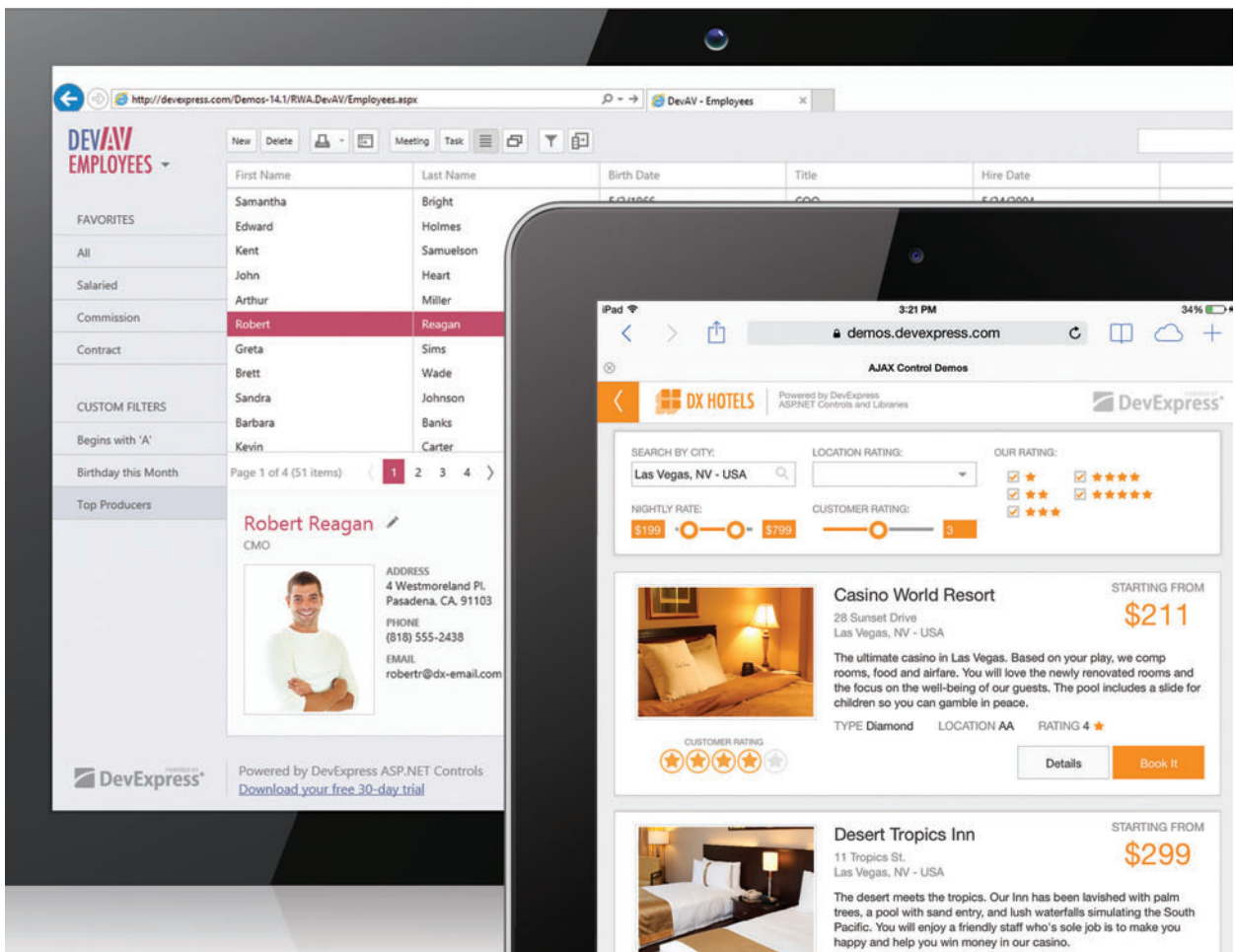
With DevExpress web controls, you can build a bridge to the future on the platform you know and love. The 175+ AJAX controls and MVC extensions that make up the DevExpress ASP.NET Subscription have been engineered so you can deliver exceptional, touch-enabled interactive experiences for the web, regardless of the target browser or computing device. DevExpress web UI components support all major browsers including Internet Explorer, FireFox, Chrome, Safari and Opera, and are continuously tested to ensure the best possible compatibility across platforms, operating systems and devices.

And so you can build your best and meet the needs of your enterprise each and every time, the DevExpress ASP.NET Subscription ships with 15 built-in application themes that can be used out-of-the box or customized via our ASP.NET Theme Builder.

“

I am a long-time DevExpress user and have been an ardent fan of the products and more importantly, the excellent support they have always provided consistently. Migrating from VCL to C# ASP.NET, I still get amazed (as a little kid) on discovering the potential of the products. Amazing suite, with support parallel-to-none!

—Dhaval Shah, Software Developer



Learn more and download your free 30-day trial
visit DevExpress.com/UISuperhero

UI Controls for Your Next Great Hybrid App



With DevExpress UI controls, you'll be able to deliver elegant line-of-business applications that emulate the touch-first Windows 8 Pro UX and maintain backward compatibility with previous versions of the Windows operating system. Whether you need to create a tile based modern UI application for Windows or need to quickly convert an existing application for use on Microsoft Surface, the DevExpress WinForms & WPF Subscriptions will help you deliver immersive business solutions, without abandoning the .NET Framework or your existing code investments.

And because we continue to extend our .NET product line with enterprise-ready capabilities designed to help you build next-generation user experiences on the desktop platform of your choice, you can rest assured that your apps will never be left behind, regardless of changes made to Windows or the introduction of new device form-factors.

“

DevExpress constantly surprises us with new features and components that we can include in our software to make it the best choice for our customers. The DevExpress support is amazing. Always quick, accurate and helpful.

—Per Olsson, Software Developer



Learn more and download your free 30-day trial
visit DevExpress.com/UISuperhero

Your Next Great Mobile App



Create highly responsive mobile apps that meet the needs of your ever changing enterprise and the BYOD world. Use the power of HTML, CSS3 and JavaScript to deliver line of business solutions that look, feel and behave just like native apps, without learning multiple languages or frameworks.

DevExpress Universal ships with the tools you'll need to deliver beautiful, easy-to-use HTML5/JavaScript applications for smartphones, tablets and desktop browsers. With nine high-impact themes replicating today's most popular mobile OS interfaces, DevExpress Mobile apps automatically apply the appropriate theme to your solution so your "write once, use everywhere" apps look great, regardless of the end-user's device or platform choice.



DevExtreme was the preferred framework for our client's mobile application... We were drawn to the fact that we could leverage our already existing HTML and JS skills to create our application, without the need to learn multiple native programming languages.

—Pierce Bizzaca, Software Developer



Your Next Great Dashboard App



With DevExpress Dashboard for .NET, creating insightful and information rich decision support systems for executives and business users across platforms and devices is a simple matter of selecting the appropriate UI element (Chart, Pivot Table, Data Card, Gauge, Map or Grid) and dropping data fields onto corresponding arguments, values, and series. And because DevExpress Dashboard automatically provides the best data visualization option for you, results are immediate, accurate and always relevant.

With full Visual Studio Integration, DevExpress Dashboard allows you to deliver business solutions that target Windows, the Web and Mobile devices with breathtaking ease. It's built so you can manage everything inside your favorite IDE: from data-binding and dashboard design to filtering, drill down and event handling.



DevExpress have allowed me to provide a fully-functional CRM, supporting marketing, sales and support activities to my end users. The CRM has full reporting, including User Report Templates and end-user designed Dashboards.

—Len Ford, Software Developer



Experience the DevExpress Difference

Award-Winning Controls and Libraries



2014

Dr. Dobb's Jolt Awards - Best Programmer Libraries

Universal Subscription

2013

DevProConnections Community Choice Awards

BEST COMPONENT SET - GOLD

Universal Subscription

BEST GRID CONTROL - GOLD

ASPxGridView and Editors Suite

BEST ADD-IN - GOLD

CodeRush for Visual Studio

BEST CHARTING & GRAPHICS TOOL - GOLD

XtraCharts Suite

BEST NAVIGATION CONTROL - GOLD

XtraBars Suite

BEST ONLINE EDITOR - GOLD

ASPxHTMLEditor Suite

BEST PRINTING/REPORTING TOOL - GOLD

Report Server

BEST SCHEDULING/CALENDAR TOOL - GOLD

ASPxScheduler Suite

BEST SHAREPOINT DEVELOPMENT TOOL - GOLD

DXperience Subscription

BEST SILVERLIGHT PRODUCT - GOLD

Silverlight Subscription

BEST TESTING/QA TOOL - GOLD

TestCafé

BEST VENDOR TECH SUPPORT - GOLD

DevExpress

2013

Visual Studio Magazine Readers Choice Awards

BEST COMPONENT SUITE, DESKTOP

DXperience Subscriptions

RUNNER-UP COMPONENT SUITE, WEB

DXperience Subscriptions

BEST GRID AND SPREADSHEET CONTROLS

XtraGrid Suite

BEST CHART, GAUGE, GRAPH AND FLOW CONTROLS

XtraCharts Suite

BEST TEXT, OCR, SCANNING & BARCODE CONTROLS

XtraRichEdit Suite

BEST UI, TOOLBAR, MENU, RIBBON CONTROLS

XtraBars Suite

BEST PDF AND PRINT/PREVIEW

XtraPrinting Library

BEST SOFTWARE DESIGN, FRAMEWORKS, AND MODELING TOOLS

eXpressApp Framework

BEST GENERAL DEVELOPMENT TOOLS

CodeRush for Visual Studio

RUNNER-UP DATABASES AND DATA DEVELOPMENT AND
MODELING

eXpress Persistent Objects

See why your peers consistently vote DevExpress #1

Visit DevExpress.com/Awards



Microsoft, Windows, Office, Outlook, Excel and Surface are trademarks or registered trademarks of Microsoft Corporation.
All other trademarks or registered trademarks are property of their respective owners.

Learn more and download your free 30-day trial
visit DevExpress.com/Try



Figure 4 Turning HRESULT Errors into Exceptions

```
struct ComException
{
    HRESULT result;

    ComException(HRESULT const value) :
        result(value)
    {}
};

void HR(HRESULT const result)
{
    if (S_OK != result)
    {
        throw ComException(result);
    }
}
```

that powers the Windows 8.1 XAML implementation to provide incredibly high-performance composition of content within a XAML application. The Internet Explorer Trident rendering engine also uses DirectComposition extensively for touch panning and zooming, as well as CSS3 animations, transitions and transforms.

I'm just going to use it to compose a swap chain that supports transparency with premultiplied alpha values on a per-pixel basis and blend it with the rest of the desktop. Traditional DirectX applications typically create a swap chain with the DXGI factory's CreateSwapChainForHwnd method. This swap chain is backed by a pair or collection of buffers that effectively would be swapped during

presentation, allowing the application to render the next frame while the previous frame is copied. The swap chain surface the application renders to is an opaque off-screen buffer. When the application presents the swap chain, DirectX copies the contents from the swap chain's back buffer to the window's redirection surface. As I mentioned earlier, the composition engine eventually composes all of the redirection surfaces together to produce the desktop as a whole.

In this case, the window has no redirection surface, so the DXGI factory's CreateSwapChainForHwnd method can't be used. However, I still need a swap chain to support Direct3D and Direct2D rendering. That's what the DXGI factory's CreateSwapChainForComposition method is for. I can use this method to create a windowless swap chain, along with its buffers, but presenting this swap chain doesn't copy the bits to the redirection surface (which doesn't exist), but instead makes it available to the composition engine directly. The composition engine can then take this surface and use it directly and in lieu of the window's redirection surface. Because this surface isn't opaque, but rather its pixel format fully supports per-pixel premultiplied alpha values, the result is pixel-perfect alpha blending on the desktop. It's also incredibly fast because there's no unnecessary copying within the GPU and certainly no copies over the bus to system memory.

That's the theory. Now it's time to make it happen. DirectX is all about the essentials of COM, so I'm going to use the Windows Runtime C++ Template Library ComPtr class template for

Figure 5 Drawing to the Swap Chain with Direct2D

```
// Create a single-threaded Direct2D factory with debugging information
ComPtr<ID2D1Factory2> d2Factory;

D2D1_FACTORY_OPTIONS const options = { D2D1_DEBUG_LEVEL_INFORMATION };

HR(D2D1CreateFactory(D2D1_FACTORY_TYPE_SINGLE_THREADED,
    options,
    d2Factory.GetAddressOf()));

// Create the Direct2D device that links back to the Direct3D device
ComPtr<ID2D1Device1> d2Device;

HR(d2Factory->CreateDevice(dxgiDevice.Get(),
    d2Device.GetAddressOf()));

// Create the Direct2D device context that is the actual render target
// and exposes drawing commands
ComPtr<ID2D1DeviceContext> dc;

HR(d2Device->CreateDeviceContext(D2D1_DEVICE_CONTEXT_OPTIONS_NONE,
    dc.GetAddressOf()));

// Retrieve the swap chain's back buffer
ComPtr<IDXGISurface2> surface;

HR(swapChain->GetBuffer(
    0, // index
    __uuidof(surface),
    reinterpret_cast<void **>(surface.GetAddressOf())));

// Create a Direct2D bitmap that points to the swap chain surface
D2D1_BITMAP_PROPERTIES1 properties = {};

properties.pixelFormat.alphaMode = D2D1_ALPHA_MODE_PREMULTIPLIED;
properties.pixelFormat.format = DXGI_FORMAT_B8G8R8A8_UNORM;
properties.bitmapOptions = D2D1_BITMAP_OPTIONS_TARGET |
    D2D1_BITMAP_OPTIONS_CANNOT_DRAW;

ComPtr<ID2D1Bitmap1> bitmap;

HR(dc->CreateBitmapFromDxgiSurface(surface.Get(),
    properties,
    bitmap.GetAddressOf()));

// Point the device context to the bitmap for rendering
dc->SetTarget(bitmap.Get());

// Draw something
dc->BeginDraw();

dc->Clear();

ComPtr<ID2D1SolidColorBrush> brush;

D2D1_COLOR_F const brushColor = D2D1::ColorF(0.18f, // red
    0.55f, // green
    0.34f, // blue
    0.75f); // alpha

HR(dc->CreateSolidColorBrush(brushColor,
    brush.GetAddressOf()));

D2D1_POINT_2F const ellipseCenter = D2D1::Point2F(150.0f, // x
    150.0f); // y

D2D1_ELLIPSE const ellipse = D2D1::Ellipse(ellipseCenter,
    100.0f, // x radius
    100.0f); // y radius

dc->FillEllipse(ellipse,
    brush.Get());

HR(dc->EndDraw());

// Make the swap chain available to the composition engine
HR(swapChain->Present(1, // sync
    0)); // flags
```


managing interface pointers. I'll also need to include and link to the DXGI, Direct3D, Direct2D and DirectComposition APIs. The following code shows you how this is done:

```
#include <wrl.h>
using namespace Microsoft::WRL;

#include <dxgi1_3.h>
#include <d3d11_2.h>
#include <d2d1_2.h>
#include <d2d1_2helper.h>
#include <dcomp.h>

#pragma comment(lib, "dxgi")
#pragma comment(lib, "d3d11")
#pragma comment(lib, "d2d1")
#pragma comment(lib, "dcomp")
```

I normally include these in my precompiled header. In that case, I'd omit the using directive and only include that in my application's source file.

I hate code samples where the error handling overwhelms and distracts from the specifics of the topic itself, so I'll also tuck this away with an exception class and an HR function to check for errors. You can find a simple implementation in **Figure 4**, but you can decide on your own error-handling policy, of course.

Now I can start to assemble the rendering stack, and that naturally begins with a Direct3D device. I'll run through this quickly because I've already described the DirectX infrastructure in detail in my March 2013 column, "Introducing Direct2D 1.1" (msdn.microsoft.com/magazine/dn198239). Here's the Direct3D 11 interface pointer:

```
ComPtr<ID3D11Device> direct3dDevice;
```

That's the interface pointer for the device, and the D3D11CreateDevice function may be used to create the device:

```
HR(D3D11CreateDevice(nullptr, // Adapter
    D3D_DRIVER_TYPE_HARDWARE,
    nullptr, // Module
    D3D11_CREATE_DEVICE_BGRA_SUPPORT,
    nullptr, 0, // Highest available feature level
    D3D11_SDK_VERSION,
    &direct3dDevice,
    nullptr, // Actual feature level
    nullptr)); // Device context
```

There's nothing too surprising here. I'm creating a Direct3D device backed by a GPU. The D3D11_CREATE_DEVICE_BGRA_SUPPORT flag enables interoperability with Direct2D. The DirectX family is held together by DXGI, which provides common GPU resource management facilities across the various DirectX APIs. Therefore, I must query the Direct3D device for its DXGI interface:

```
ComPtr<IDXGIDevice> dxgiDevice;
HR(direct3dDevice.As(&dxgiDevice));
```

The ComPtr As method is just a wrapper for the QueryInterface method. With the Direct3D device created, I can then create the swap chain that will be used for composition. To do that, I first need to get a hold of the DXGI factory:

```
ComPtr<IDXGIFactory2> dxFactory;

HR(CreateDXGIFactory2(
    DXGI_CREATE_FACTORY_DEBUG,
    __uuidof(dxFactory),
    reinterpret_cast<void **>(dxFactory.GetAddressOf())));
```

Here, I'm opting for extra debugging information—an invaluable aid during development. The hardest part of creating a swap chain is figuring out how to describe the desired swap chain to the DXGI factory. This debugging information is immensely helpful in fine-tuning the necessary DXGI_SWAP_CHAIN_DESC1 structure:

```
DXGI_SWAP_CHAIN_DESC1 description = {};
```

This initializes the structure to all zeroes. I can then begin to fill in any interesting properties:

```
description.Format = DXGI_FORMAT_B8G8R8A8_UNORM;
description.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
description.SwapEffect = DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL;
description.BufferCount = 2;
description.SampleDesc.Count = 1;
description.AlphaMode = DXGI_ALPHA_MODE_PREMULTIPLIED;
```

The particular format, a 32-bit pixel format with 8 bits for each color channel along with a premultiplied 8-bit alpha component, isn't your only option, but provides the best performance and compatibility across devices and APIs.

The swap chain's buffer usage must be set to allow render target output to be directed to it. This is necessary so the Direct2D device context can create a bitmap to target the DXGI surface with drawing commands. The Direct2D bitmap itself is merely an abstraction backed by the swap chain.

Composition swap chains only support the flip-sequential swap effect. This is how the swap chain relates to the composition engine in lieu of a redirection surface. In the flip model, all buffers are shared directly with the composition engine. The composition engine can then compose the desktop directly from the swap chain back buffer without additional copying. This is usually the most efficient model. It's also required for composition, so that's what I use. The flip model also necessarily requires at least two buffers, but doesn't support multisampling, so BufferCount is set to two and SampleDesc.Count is set to one. This count is the number of multisamples per pixel. Setting this to one effectively disables multisampling.

Finally, the alpha mode is critical. This would normally be ignored for opaque swap chains, but in this case I really do want transparency behavior to be included. Premultiplied alpha values typically provide the best performance, and it's also the only option supported by the flip model.

The final ingredient before I can create the swap chain is to determine the desired size of the buffers. Normally, when calling the CreateSwapChainForHwnd method, I can ignore the size and the DXGI factory will query the window for the size of the client area. In this case, DXGI has no idea what I plan to do with the swap chain, so I need tell it specifically what size it needs to be. With the window created, this is a simple matter of querying the window's client area and updating the swap chain description accordingly:

```
RECT rect = {};
GetClientRect(window, &rect);
```

```
description.Width = rect.right - rect.left;
description.Height = rect.bottom - rect.top;
```

I can now create the composition swap chain with this description and create a pointer to the Direct3D device. Either the Direct3D or DXGI interface pointers may be used:

```
ComPtr<IDXGISwapChain1> swapChain;

HR(dxFactory->CreateSwapChainForComposition(dxgiDevice.Get(),
    &description,
    nullptr, // Don't restrict
    swapChain.GetAddressOf()));
```

Now that the swap chain is created, I can use any Direct3D or Direct2D graphics rendering code to draw the application, using alpha values as needed to create the desired transparency. There's nothing new here, so I'll refer you to my March 2013 column again for the specifics of rendering to a swap chain with Direct2D.

Figure 5 provides a simple example if you're following along. Just don't forget to support per-monitor DPI awareness as I described in my February 2014 column, "Write High-DPI Apps for Windows 8.1" (msdn.microsoft.com/magazine/dn574798).

Now I can finally begin to use the DirectComposition API to bring it all together. While the Windows composition engine deals with rendering and composition of the desktop as a whole, the DirectComposition API allows you to use this same technology to compose the visuals for your applications. Applications compose together different elements, called visuals, to produce the appearance of the application window itself. These visuals can be animated and transformed in a variety of ways to produce rich and fluid UIs. The composition process itself is also performed along with the composition of the desktop as a whole, so more of your application's presentation is taken off your application threads for improved responsiveness.

DirectComposition is primarily about composing different bitmaps together. As with Direct2D, the concept of a bitmap here is more of an abstraction to allow different rendering stacks to cooperate and produce smooth and appealing application UXes.

Like Direct3D and Direct2D, DirectComposition is a DirectX API that's backed and powered by the GPU. A DirectComposition device is created by pointing back to the Direct3D device, in much the same way a Direct2D device points back to the underlying Direct3D device. I use the same Direct3D device I previously used to create the swap chain and Direct2D render target to create a DirectComposition device:

```
ComPtr<IDCompositionDevice> dcompDevice;

HR(DCompositionCreateDevice(
    dxgiDevice.Get(),
    __uuidof(dcompDevice),
    reinterpret_cast<void **>(dcompDevice.GetAddressOf())));
```

The DCompositionCreateDevice function expects the Direct3D device's DXGI interface and returns an IDCompositionDevice interface pointer to the newly created DirectComposition device. The DirectComposition device acts as a factory for other DirectComposition objects and provides the all-important Commit

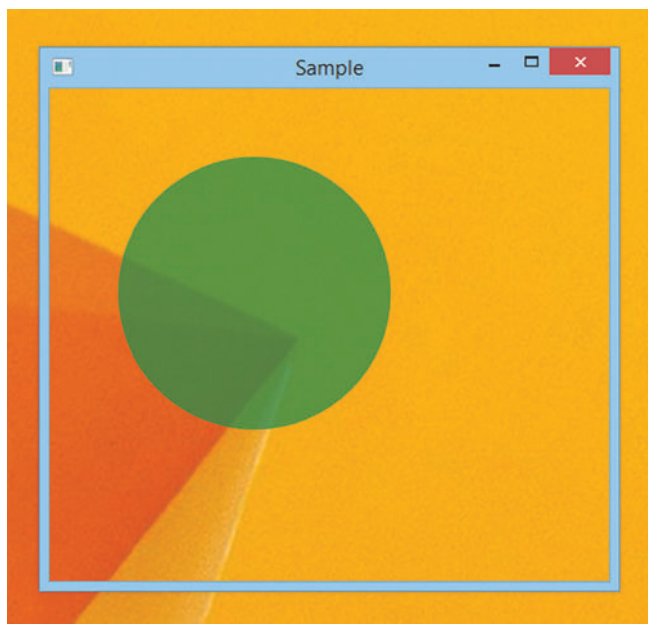


Figure 6 Direct2D Drawing on a DirectComposition Surface

method that commits the batch of rendering commands to the composition engine for eventual composition and presentation.

Next up, I need to create a composition target that associates the visuals that will be composed with the destination, which is the application window:

```
ComPtr<IDCompositionTarget> target;

HR(dcompDevice->CreateTargetForHwnd(window,
                                     true, // Top most
                                     target.GetAddressOf()));
```

The CreateTargetForHwnd method's first parameter is the window handle returned by the CreateWindowEx function. The second parameter indicates how the visuals will be combined with any other window elements. The result is an IDCompositionTarget interface pointer whose only method is called SetRoot. It lets me set the root visual in a possible tree of visuals to be composed together. I don't need a whole visual tree, but I need at least one visual object, and for that I can once again turn to the DirectComposition device:

```
ComPtr<IDCompositionVisual> visual;
HR(dcompDevice->CreateVisual(visual.GetAddressOf()));
```

The visual contains a reference to a bitmap and provides a set of properties that affect how that visual will be rendered and composed relative to other visuals in the tree and the target itself. I already have the content I want this visual to carry forward to the composition engine. It's the swap chain I created earlier:

```
HR(visual->SetContent(swapChain.Get()));
```

The visual is ready and I can simply set it as the root of the composition target:

```
HR(target->SetRoot(visual.Get()));
```

Finally, once the shape of the visual tree has been established, I can simply inform the composition engine that I'm done by calling the Commit method on the DirectComposition device:

```
HR(dcompDevice->Commit());
```

For this particular application, where the visual tree doesn't change, I only need to call Commit once at the beginning of the application and never again. I originally assumed the Commit method needed to be called after presenting the swap chain, but this isn't the case because swap chain presentation isn't synchronized with changes to the visual tree.

Figure 6 shows what the application window looks like now that Direct2D has rendered to the swap chain and DirectComposition has provided the partly transparent swap chain to the composition engine.

I'm excited to finally have a solution for an old problem: the ability to produce high-performance windows that are alpha blended with the rest of the desktop. I'm excited about the possibilities the DirectComposition API enables and what it means for the future of application UX design and development in native code.

Want to draw your own window chrome? No problem; just replace the WS_OVERLAPPEDWINDOW window style with the WS_POPUP window style when creating the window. Happy coding! ■

KENNY KERR is a computer programmer based in Canada, as well as an author for Pluralsight and a Microsoft MVP. He blogs at kennykerr.ca and you can follow him on Twitter at twitter.com/kennykerr.

THANKS to the following technical expert for reviewing this article:
Leonardo Blanco (Microsoft)



Telemetry Ingestion and Analysis Using Microsoft Azure Services

[The regular columnists Bruno Terkaly and Ricardo Villalobos present a guest columnist for this month's column. They will return with the next installment. —Ed.]

Every sensor-based device generates telemetry data. Interpreting this data is at the core of its value proposition. In the consumer world, a driver looks at his connected car dashboard to see how his driving style affects fuel consumption and traffic. In the industrial world, comparing the temperature of a machine to the average of others on the factory floor can help an operator identify the risks of failure and perform predictive maintenance.

You need to analyze this data to provide meaningful visualizations and insights.

These scenarios require telemetry data from tens or hundreds of thousands of connected devices. What's more important, you need to analyze this data to provide meaningful visualizations and insights. When dealing with such large amounts of data, Big Data frameworks such as Hadoop build a solid data-processing foundation that can scale up with the installed base of devices.

In this article, you'll learn how to create a simple telemetry ingestion architecture using the Microsoft Azure Service Bus. Then you'll consume and analyze this data in a scalable way using the Microsoft Azure Hadoop service called HDInsight.

Solution Architecture

In previous columns, Bruno Terkaly and Ricardo Villalobos showed how to use the Service Bus to establish a command channel to communicate with a connected object. In this article, I'll use the Service Bus as a middleware layer to buffer telemetry messages sent by the device.

The devices will communicate directly with the Service Bus to send telemetry messages to a dedicated Topic (see **Figure 1**). Then one

or several subscriptions will de-queue these messages in a worker role and store them as flat files in Blob storage. The Hadoop cluster can then use these input files to perform analysis and calculations.

This architecture has the benefit of decoupling the various pieces from each other. The Service Bus acts as middleware and can buffer data if the workers can't read them fast enough. You can monitor the queue length and use that as the basis for auto-scaling the worker tier.

The subscriptions are also useful for performing simple filtering on incoming data and routing it to different back-end processing tiers. For example, you could have an Urgent subscription that sends messages to a real-time alerting system, and use an Everything subscription to capture all data for later analysis.

Because the workers just move data into storage—whether Hadoop Distributed File System (HDFS) or Blob storage—it's decoupled from the Hadoop processing piece. This can run independently of the incoming data rhythm. You could choose to have a Hadoop cluster running permanently. This lets you process small batches all the time and reduce computational latency. You could also choose to save money by having an HDInsight cluster start just once per day to perform all calculations in one batch. You can also have a mix of the two.

Ingest Telemetry Data Using the Service Bus

The Azure Service Bus offers two protocol choices to send messages to a Topic: HTTP or AMQP. In the case of connected devices, often with limited bandwidth, AMQP has some advantages. It's an efficient, binary, reliable and portable protocol. It also has libraries for many languages, runtime environments and OSes. This gives you flexibility when connecting your device directly to the Service Bus to send telemetry messages.

To test this approach, I used a Raspberry Pi board to feed temperature and other sensor data, using the Apache Qpid Proton AMQP library. Proton is a bare-bones, portable library you can compile on a variety of environments to send AMQP messages. It's completely interoperable with the Azure Service Bus. Find more information about the Proton AMQP library at bit.ly/1icc6Ag.



Figure 1 Basic Flow of Big Data Telemetry Solution

For this example, I've compiled the Proton library directly on the Raspberry Pi board. I used the Python bindings to write a simple script to capture sensor readings from the USB serial port and send them to the Azure Service Bus, which you can see in **Figure 2**.

The Python script directly addresses the Azure Service Bus Topic named "telemetry." It's using a connection string that includes the standard Service Bus authentication token and specifies using the AMQP protocol. In a real-world environment, you'd need to use a more sophisticated authentication mechanism to ensure your connection parameters aren't compromised.

You have several choices for which type of Hadoop solution you'll use for data analysis.

Assume a significant number of these Raspberry devices start gathering data. Each one will send a Device ID (DID) you'll use again later to calculate average temperatures. In this example, the DID is generated with the UUID module to retrieve the system's MAC address.

An Arduino Esplora board connected to the Raspberry Pi via USB gathers the readings. The Esplora is an all-in-one board with

Figure 2 Python Code in the Raspberry Pi Reading to Capture Sensor Readings

```
#!/usr/bin/python

import sys
import commands
import re
import uuid
import serial
from proton import *

# Device ID
id = uuid.getnode()

# Topic address
address = "amqps://owner:key@address.servicebus.windows.net/telemetry"

# Open serial port
ser = serial.Serial('/dev/ttyACM0', 9600)

# Create Proton objects
messenger = Messenger()

while True:
    # Read values from Arduino in the form K1:V1_K2:V2_...
    temp = ser.readline().rstrip('\r\n')
    print temp

    # Create AMQP message
    message = Message()

    # Initialize properties
    message.properties = dict()
    message.properties[symbol("did")] = symbol(id)

    # Map string to list, symbolize, create dict and merge
    pairs=map(lambda x:x.split(':'), temp.split('_'))
    symbols = map(lambda x:(symbol(x[0]),int(x[1])), pairs)
    message.properties.update(dict(symbols))

    message.address = address
    messenger.put(message)
    messenger.send()
```

built-in sensors. That makes it easy to read temperature or other environmental parameters and send them to the serial bus. The Python script at the other end of the USB cable then reads the output values. An example of an Arduino schema that prints sensor values to the serial port is shown in **Figure 3**.

Select Your Big Data Deployment

You have several choices for which type of Hadoop solution you'll use for data analysis. The choice of deployment type will dictate how and where you'll need to aggregate data for analysis.

Azure offers a compelling solution with HDInsight. This exposes the Hadoop framework as a service. This distribution of Hadoop, based on the Hortonworks Data Platform (HDP) for Windows, comes with a connector that lets jobs directly access input data from Azure Blob storage.

This means you don't have to have the Hadoop cluster up and running in order to receive input files. You can upload files to a Blob storage container that HDInsight will use later. When you analyze a batch of files, you can start the HDInsight cluster in a few minutes, execute a series of jobs for a couple of hours and then shut it down. This translates into lower bills in terms of compute hours.

On the other hand, if you choose to deploy a standard Hadoop distribution such as HDP, or the Cloudera Distribution on Azure virtual machines (VMs), you'll be responsible for keeping the cluster up-to-date. You'll also have to have it properly configured for optimal operation. This approach makes sense if you intend to use custom Hadoop components not included in HDInsight, such as HBase, as the storage mechanism.

Save Telemetry Data to Blob Storage

Extracting data from the Azure Service Bus is a simple process. Use a worker role as a "reader" or "listener" for the subscription. Then, accumulate messages into input files HDInsight can use.

First, set up one or several subscriptions on your Azure Service Bus Topic. This gives you some latitude when splitting or routing the data stream, depending on the requirements. At the very least, it's a good idea to create a "catch-all" subscription to store all

Figure 3 Arduino Code Gathering Raspberry Pi Readings

```
void loop()
{
    int celsius = Esplora.readTemperature(DEGREES_C);
    int loudness = Esplora.readMicrophone();
    int light = Esplora.readLightSensor();

    Serial.print("T:");
    Serial.print(celsius);

    Serial.print("_");

    Serial.print("M:");
    Serial.print(loudness);

    Serial.print("_");

    Serial.print("L:");
    Serial.print(light);

    Serial.println();

    // Wait a second
    delay(1000);
}
```

Figure 4 An Azure Service Bus Subscription

```
var namespaceManager = NamespaceManager.CreateFromConnectionString(connectionString);

// Create the Topic
if (!namespaceManager.TopicExists("telemetry"))
{
    namespaceManager.CreateTopic("telemetry");
}

// Create a "catch-all" Subscription
if (!namespaceManager.SubscriptionExists("telemetry", "all"))
{
    namespaceManager.CreateSubscription("telemetry", "all");
}

// Create an "alerts" subscription
if (!namespaceManager.SubscriptionExists("telemetry", "alert"))
{
    SqlFilter alertFilter = new SqlFilter("type = 99");
    namespaceManager.CreateSubscription("telemetry", "alert", alertFilter);
}
```

incoming messages. You can also use Filters on Azure Service Bus subscriptions. This will create additional streams for specific messages. An example of creating the Topic and Subscriptions using C# and the Azure Service Bus SDK library is shown in **Figure 4**.

Once you've created the Azure Service Bus Subscription, you can receive and save messages. This example uses the CSV format, which is easy to read and understand both by machines and humans. To read the incoming messages as quickly as possible, the worker creates a number of Tasks (there are 10 in this example). It also uses Async methods to read batches of messages, instead of reading them one at a time. The "all" Subscription and "telemetry" topic will receive the messages (see **Figure 5**).

The `TelemetryMessage.Stringify` method simply returns a line of text in CSV format that contains the telemetry data. It can also extract some useful fields from the Azure Service Bus headers, such as the Message ID or the Enqueued Time.

Figure 5 Receiving Messages from the Subscription and Storing Them in Blob Storage

```
SubscriptionClient client = SubscriptionClient.
CreateFromConnectionString(connectionString, "telemetry", "all",
ReceiveMode.ReceiveAndDelete);
List<Task> tasks = new List<Task>();
for (int i = 0; i < NBTASKS; i++)
{
    var id = i; // Closure alert
    Task t = Task.Run(async () =>
    {
        BlobStorageWriter writer = new BlobStorageWriter(id);
        while (true)
        {
            var messages = await client.ReceiveBatchAsync(BATCH_SIZE);
            foreach (var message in messages)
            {
                try
                {
                    await writer.WriteLine(TelemetryMessage.Stringify(message));
                }
                catch (Exception ex)
                {
                    Trace.TraceError(ex.Message);
                }
            }
        }
    });
    tasks.Add(t);
}
Task.WaitAll(tasks.ToArray());
```

The job of `BlobStorageWriter.WriteLine` is to write the line directly into a Blob. Because 10 tasks are available in parallel, that same number of Blobs will be affected at once. `WriteOneLine` also rotates files from time to time for HDInsight to pick them up. I use two parameters to decide when to switch to a new file: the number of lines written to the file and the time since the Blob was created (for example, creating a new file every hour or when it reaches 1,000,000 lines). This method uses asynchronous calls to avoid blocking while writing messages to the Blob stream (see **Figure 6**).

The resulting files contain data extracted from the telemetry messages, as shown:

```
145268284e8e498282e20b01170634df,test,24,980,21,2014-03-14 13:43:32
dbb52a3cf690467d8401518fc5e266fd,test,24,980,21,2014-03-14 13:43:32
e9b5f508ef8c4d1e8d246162c02e7732,test,24,980,21,2014-03-14 13:43:32
```

They include the Message ID, Device ID, three of the readings and the date the message was enqueued. This format is easy to parse in the next step.

Analyze the Data Using HDInsight

The most impressive benefit of HDInsight is that you can start a complete Hadoop cluster, run a job and deprovision the cluster directly from the command line. You don't ever have to log on to a VM or perform any custom configuration. You can provision and manage HDInsight with Windows PowerShell on Windows, or using cross-platform command-line tools on Mac or Linux.

When you analyze a batch of files, you can start the HDInsight cluster in a few minutes, execute a series of jobs for a couple of hours and then shut it down.

You can download the integrated Azure PowerShell commandlets from bit.ly/1tGirZk. These commandlets include everything you need to manage your Azure infrastructure, including HDInsight clusters. Once you've imported your publishing settings and selected the default subscription, you only need one command line to create a new HDInsight cluster:

```
New-AzureHDInsightCluster -Name "hditelometry" -Location "North
Europe" -DefaultStorageAccountName "telemetry.blob.core.
windows.net" -DefaultStorageAccountKey "storage-account-key"
-DefaultStorageContainerName "data" -ClusterSizeInNodes 4
```

This command instructs the HDInsight cluster to use the existing Storage Account and Container as its file system root. This is how it will access all the telemetry data the ingestion process generates. You can also select how many worker nodes the cluster should use, depending on the volume of data and how much parallelism you need.

Once the cluster is up and running, you can enable Remote Desktop access. Doing so lets other users log on to the head node to start an interactive session with standard Hadoop commands and tools. However, it's much faster to use remote commands,

We didn't invent the Internet...

...but our components help you power the apps that bring it to business.



TOOLS • COMPONENTS • ENTERPRISE ADAPTERS

- **E-Business**
AS2, EDI/X12, NAESB, OFTP ...
- **Credit Card Processing**
Authorize.Net, TSYS, FDMS ...
- **Shipping & Tracking**
FedEx, UPS, USPS ...
- **Accounting & Banking**
QuickBooks, OFX ...
- **Internet Business**
Amazon, eBay, PayPal ...
- **Internet Protocols**
FTP, SMTP, IMAP, POP, WebDav ...
- **Secure Connectivity**
SSH, SFTP, SSL, Certificates ...
- **Secure Email**
S/MIME, OpenPGP ...
- **Network Management**
SNMP, MIB, LDAP, Monitoring ...
- **Compression & Encryption**
Zip, Gzip, Jar, AES ...



The Market Leader in Internet Communications, Security, & E-Business Components

Each day, as you click around the Web or use any connected application, chances are that directly or indirectly some bits are flowing through applications that use our components, on a server, on a device, or right on your desktop. It's your code and our code working together to move data, information, and business. We give you the most robust suite of components for adding Internet Communications, Security, and E-Business Connectivity to

any application, on any platform, anywhere, and you do the rest. Since 1994, we have had one goal: to provide the very best connectivity solutions for our professional developer customers. With more than 100,000 developers worldwide using our software and millions of installations in almost every Fortune 500 and Global 2000 company, our business is to connect business, one application at a time.

connectivity
powered by 

To learn more please visit our website →

www.nsoftware.com

Figure 6 Write Data from Messages to Azure Blobs

```
public async Task WriteOneLine(string line)
{
    var bytes = Encoding.UTF8.GetBytes(string.Format("{0}\n", line));
    await destinationStream.WriteAsync(bytes, 0, bytes.Length);
    TimeSpan ts = DateTime.Now - startBlobTime;
    if (++linesWritten > MAX_LINES || ts.TotalSeconds > MAX_SECONDS)
    {
        Trace.TraceInformation(
            "Wrote " + linesWritten + " lines to " + currentBlob.Name);
        GetNextBlob();
        linesWritten = 0;
    }
}
```

taking advantage of Windows PowerShell to launch Map Reduce, Hive or Pig jobs.

I used a Pig job to calculate the average temperature value. Pig was initially developed at Yahoo. It lets people using Hadoop focus more on analyzing large data sets and spend less time writing mapper and reducer programs. A Pig script typically has three stages:

1. Load the data you want to manipulate.
2. Run a series of data transformations (which are translated into a set of mapper and reducer tasks).
3. Dump the results to screen, or store the results in a file.

The most impressive benefit of Azure HDInsight is that you can start a complete Hadoop cluster, run a job and deprovision the cluster directly from the command line.

The following example shows how you typically achieve this by running the script interactively, in an Exploratory Data Analysis (EDA) phase, with the Pig interpreter:

```
data = load '/telemetry*.csv' using PigStorage(',') as (id:chararray,
did:chararray, temp:int, light:int, mic:int, timestamp:datetime);
data1 = group data by did;
data2 = foreach data1 generate group as did, COUNT(data), AVG(data.temp);
dump data2;
```

If you type this script directly into the Pig interpreter, it will display a table containing the number of temperature data points and the average measured value for each DID. As you can see, the Pig syntax is quite explicit. The different data manipulation steps are clearly separated:

- The first load statement is used to load the data from the CSV files, describing the name and types of the input fields.
- The data is then grouped by DID, or per device.
- The result dataset is generated with aggregate functions like COUNT and AVG.

Once the script is finalized, you can automate this task with Windows PowerShell. Use the `New-AzureHDInsightPigJob-Definition` commandlet to initialize a Pig job with the script created.

Figure 7 Insert, Analyze and Start Jobs in HDInsight

```
$PigScript = "data = load '/telemetry*.csv' using PigStorage(',') as (id:chararray,
did:chararray, temp:int, light:int, mic:int, timestamp:datetime);" +
"data1 = group data by did;" +
"data2 = foreach data1 generate group as did, COUNT(data), AVG(data.temp);" +
"dump data2;"

# Define a Pig job
$PigJobDefinition = New-AzureHDInsightPigJobDefinition -Query $PigScript

# Start the job
$PigJob = Start-AzureHDInsightJob -Cluster "hditelemetry" -JobDefinition
$PigJobDefinition

# Wait for the job to finish
Wait-AzureHDInsightJob -Job $PigJob -WaitTimeoutInSeconds 3600

# Get the job results
Get-AzureHDInsightJobOutput -Cluster "hditelemetry" -JobId $PigJob.JobId
-StandardOutput
```

Then you can use `Start-AzureHDInsightJob` and `Wait-AzureHDInsightJob` to start the job and wait for its conclusion (see Figure 7). You can then use `Get-AzureHDInsightJobOutput` to retrieve the results.

The displayed result in the command-line console looks like this:

```
C:\> Get-AzureHDInsightJobOutput -Cluster "hditelemetry" -JobId $PigJob.JobId
(test,29091,24.0)
(49417795060,3942,30.08371385083714)
```

In this case, there are quite a few test measurements and around 4,000 readings from the Raspberry Pi. The readings average 30 degrees.

Wrapping Up

The Azure Service Bus is a reliable and fast way to gather data from all sorts of devices. In order to store and analyze that data, you need a robust storage and analysis engine. Azure HDInsight abstracts the process of creating and maintaining a Hadoop cluster for this degree of storage. It's a highly scalable solution you can configure and automate using tools such as Windows PowerShell or the Mac/Linux Azure command-line interface. ■

THOMAS CONTE is a technical evangelist for the Microsoft Azure platform in the Developer & Platform Evangelism (DPE) division. His role is to facilitate access to technology for developers, architects, and software partners through code samples, publications, and public speaking. He endeavors to run Microsoft Azure on as many non-Microsoft technologies from the open source world as possible. Follow him at twitter.com/tomconte.

BRUNO TERKALY is a developer evangelist for Microsoft. His depth of knowledge comes from years of experience in the field, writing code using a multitude of platforms, languages, frameworks, SDKs, libraries and APIs. He spends time writing code, blogging and giving live presentations on building cloud-based applications, specifically using the Microsoft Azure platform. You can read his blog at blogs.msdn.com/b/brunoterkaly.

RICARDO VILLALOBOS is a seasoned software architect with more than 15 years of experience designing and creating applications for companies in multiple industries. Holding different technical certifications, as well as a master's degree in business administration from the University of Dallas, he works as a cloud architect in the DPE Globally Engaged Partners team for Microsoft, helping companies worldwide to implement solutions in Microsoft Azure. You can read his blog at blog.ricardovillalobos.com.

THANKS to the following Microsoft technical experts for reviewing this article: Rafael Godinho and Jeremiah Talkar



BEST SELLER

ComponentOne Studio Enterprise 2014 v1

from **\$1,315.60**



.NET Tools for the Professional Developer: Windows, HTML5/Web, and XAML.

- Hundreds of UI controls for all .NET platforms including grids, charts, reports and schedulers
- Visual Studio 2013 and Bootstrap support
- Advanced theming tools for WinForms and ASP.NET
- 40+ UI widgets built with HTML5, jQuery, CSS3, and SVG
- Windows Store Sparkline, DropDown, & Excel controls



BEST SELLER

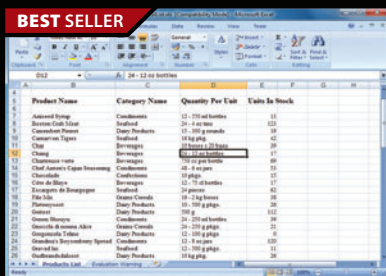
Help & Manual Professional

from **\$583.10**



Easily create documentation for Windows, the Web and iPad.

- Powerful features in an easy accessible and intuitive user interface
- As easy to use as a word processor, but with all the power of a true WYSIWYG XML editor
- Single source, multi-channel publishing with conditional and customized output features
- Output to HTML, WebHelp, CHM, PDF, ePUB, RTF, e-book or print
- Styles and Templates give you full design control



BEST SELLER

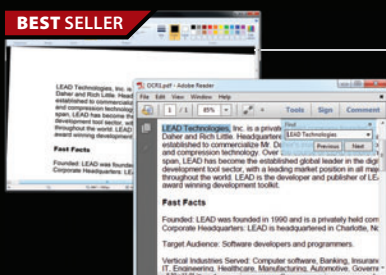
Aspose.Total for .NET

from **\$2,449.02**



Every Aspose .NET component in one package.

- Programmatically manage popular file formats including Word, Excel, PowerPoint and PDF
- Work with charts, diagrams, images, project plans, emails, barcodes, OCR and OneNote files alongside many more document management features in .NET applications
- Common uses also include mail merging, adding barcodes to documents, building dynamic reports on the fly and extracting text from PDF files



BEST SELLER

LEADTOOLS Document Imaging SDKs V18

from **\$2,695.50**



Add powerful document imaging functionality to desktop, tablet, mobile & web applications.

- Comprehensive document image cleanup, preprocessing, viewer controls and annotations
- Fast and accurate OCR, OMR, ICR and Forms Recognition with multi-threading support
- PDF & PDF/A Read / Write / Extract / View / Edit
- Barcode Detect / Read / Write for UPC, EAN, Code 128, Data Matrix, QR Code, PDF417
- Zero-Footprint HTML5/JavaScript Controls & Native WinRT Libraries for Windows Store

Enhance Your JavaScript Investment with TypeScript

Bill Wagner

The **TypeScript** programming language is actually a proper superset of JavaScript. If you're using JavaScript, you're already writing TypeScript. That doesn't mean you're writing good TypeScript and making use of all its features. It does mean you have a smooth migration path from your existing JavaScript investment to a TypeScript codebase that leverages the new features TypeScript offers.

In this article, I'll provide recommendations on migrating an application from JavaScript to TypeScript. You'll learn how to move from JavaScript to TypeScript to use the TypeScript type system to help you write better code. With TypeScript static analysis, you'll minimize errors and be more productive. By following these recommendations, you'll also minimize the amount of errors and warnings from the TypeScript type system during the migration.

I'll start with an application that manages an address book as an example. It's a Single-Page Application (SPA) using JavaScript

on the client. I've kept it simple for this article and only included the portion that displays a list of contacts. It uses the Angular framework for data binding and application support. The Angular framework handles the data binding and templating for displaying the contact information.

Three JavaScript files make up the application: The `app.js` file contains the code that starts the app. The `contactsController.js` file is the controller for the listing page. The `contactsData.js` file contains a list of contacts that will be displayed. The controller—along with the Angular framework—handles the behavior of the listing page. You can sort the contacts and show or hide contact details for any single contact. The `contactsData.js` file is a hardcoded set of contacts. In a production application, this file would contain code to call a server and retrieve data. The hardcoded list of contacts makes a more self-contained demo.

Don't worry if you don't have much experience with Angular. You'll see how easy it is to use as I start migrating the application. The application follows Angular conventions, which are easy to preserve as you migrate an application to TypeScript.

The best place to begin migrating an application to TypeScript is with the controller file. Because any valid JavaScript code is also valid TypeScript code, simply change the extension of the controller file called `contactsController.js` from `.js` to `.ts`. The TypeScript language is a first-class citizen in Visual Studio 2013 Update 2. If you have the Web Essentials extension installed, you'll see both the TypeScript source and the generated JavaScript output in the same window (see **Figure 1**).

This article discusses:

- Making a smooth migration from JavaScript to TypeScript
- How to leverage the full feature set of TypeScript
- Using the Angular framework to ensure a smooth migration

Technologies discussed:

JavaScript, TypeScript, Visual Studio 2013 Update 2

Code download available at:

msdn.microsoft.com/magazine/msdnmag0614

Because TypeScript-specific language features aren't being used yet, those two views are almost the same. The extra comment line at the end provides information for Visual Studio when debugging TypeScript applications. Using Visual Studio, an application can be debugged at the TypeScript level, instead of the generated JavaScript source level.

You can see the TypeScript compiler reports an error for this application, even though the compiler generates valid JavaScript output. That's one of the great features of the TypeScript language. It's a natural consequence of the rule that TypeScript is a strict superset of JavaScript. I haven't declared the symbol `contactsApp` in any TypeScript file yet. Therefore, the TypeScript compiler assumes the type of `any`, and assumes that symbol will reference an object at run time. Despite those errors, I can run the application and it will still work correctly.

I could continue and change the extensions of all the JavaScript files in the application. But I wouldn't recommend doing that just yet, because there will be a lot more errors. The application will still work, but having so many errors makes it harder to use the TypeScript system to help you write better code. I prefer working on one file at a time, and adding type information to the application as I go. That way I have a smaller number of type system errors to fix at once. After I have a clean build, I know the TypeScript compiler is helping me avoid those mistakes.

It's easy to declare an external variable for the `contactsApp`. By default, it would have the `any` type:

```
declare var contactsApp: any;
```

While that fixes the compiler error, it doesn't help avoid mistakes when calling methods in the Angular library. The `any` type is just what it sounds like: It could be anything. TypeScript won't perform any type checking when you access the `contactsApp` variable. To get type checking, you need to tell TypeScript about the type of `contactsApp` and about the types defined in the Angular framework.

TypeScript enables type information for existing JavaScript libraries with a feature called Type Definitions. A Type Definition is a set of declarations with no implementation. They describe the types and their APIs to the TypeScript compiler. The Definitely-Typed project on GitHub has type definitions for many popular JavaScript libraries, including Angular.js. I include those definitions in the project using the NuGet package manager.

Once the Type Definitions for the Angular library have been included, I can use them to fix the compiler errors I'm seeing. I need to reference the type information I just added to the project. There's a special comment that tells the TypeScript compiler to reference type information:

```
/// <reference path="../../Scripts/typings/angularjs/angular.d.ts" />
```

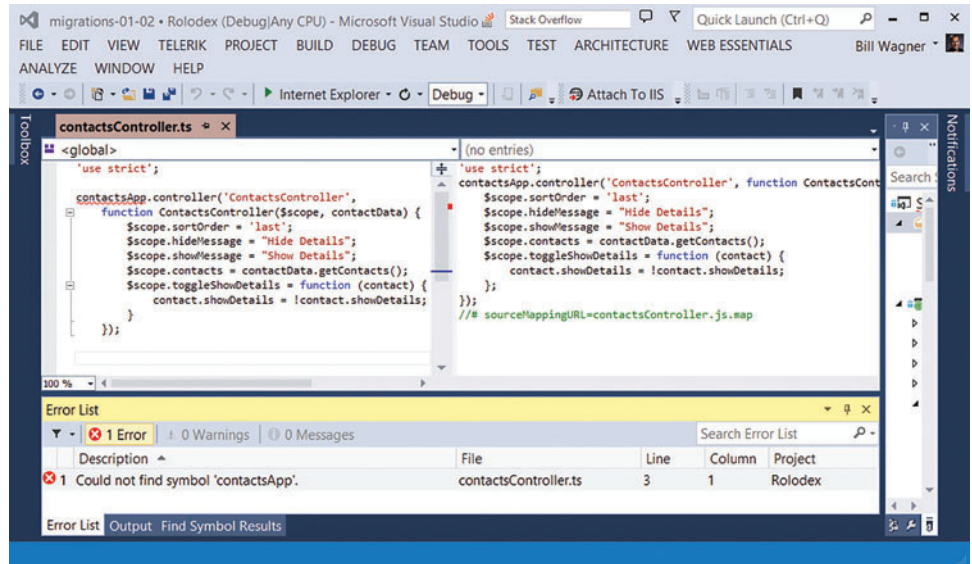


Figure 1 The TypeScript Editing Experience in Visual Studio 2013 Update 2

The TypeScript compiler can now interpret any of the types defined in the Type Definition file `angular.d.ts`. It's time to fix the type of the `contactsApp` variable. The expected type of the `contactsApp` variable, which is declared in `app.js` in the `ng` namespace, is an `IModule`:

```
declare var contactsApp: ng.IModule;
```

With this declaration, I'll get IntelliSense whenever a period is pressed after `contactsApp`. I'll also get error reports from the TypeScript compiler whenever I mistype or misuse the APIs declared on the `contactsApp` object. The compiler errors are gone and I've included static type information for the app object.

The rest of the code in the `contactsController` object still lacks type information. Until you add type annotations, the TypeScript compiler will assume any variable is of the `any` type. The second parameter to the `contactsApp.controller` method is a function and that function's first parameter, `$scope`, is of type `ng.IScope`. So I'll include that type on the function declaration (`contactData` will still be interpreted as the `any` type):

```
contactsApp.controller('ContactsController',
function ContactsController($scope: ng.IScope, contactData) {
    $scope.sortOrder = 'last';
    $scope.hideMessage = "Hide Details";
    $scope.showMessage = "Show Details";
    $scope.contacts = contactData.getContacts();
    $scope.toggleShowDetails = function (contact) {
        contact.showDetails = !contact.showDetails;
    }
});
```

Figure 2 Including Properties on a Contact Object

```
interface IContact {
    first: string;
    last: string;
    address: string;
    city: string;
    state: string;
    zipCode: number;
    cellPhone: number;
    homePhone: number;
    workPhone: number;
    showDetails: boolean
}
```

This introduces a new set of compiler errors. The new errors are because the code inside that `contactsController` function manipulates properties that aren't part of the `ng.IScope` type. `Ng.IScope` is an interface, and the actual `$scope` object is an application-specific type that implements `IScope`. Those properties being manipulated are members of that type. To leverage the TypeScript static typing, I need to define that application-specific type. I'll call it `IContactsScope`:

```
interface IContactsScope extends ng.IScope {
    sortOrder: string;
    hideMessage: string;
    showMessage: string;
    contacts: any;
    toggleShowDetails: (contact: any) => boolean;
}
```

Once the interface is defined, I simply change the type of the `$scope` variable in the function declaration:

```
function ContactsController($scope : IContactsScope, contactData) {
```

After making these changes, I can build the app without errors and it will run correctly. There are several important concepts to observe when adding this interface. Notice I didn't have to find any other code and declare that any particular type implements the `IContactsScope` type. TypeScript supports structural typing, colloquially referred to as "duck typing." This means any object that declares the properties and methods declared in `IContactsScope` implements the `IContactsScope` interface, whether or not that type declares that it implements `IContactsScope`.

Notice I'm using the `any` TypeScript type as a placeholder in the definition of `IContactsScope`. The `contacts` property represents the list of contacts and I haven't migrated the `Contact` type yet. I can use `any` as a placeholder and the TypeScript compiler won't perform any type checking on access to those values. This is a useful technique throughout an application migration.

The `any` type represents any types I haven't yet migrated from JavaScript to TypeScript. It makes the migration go more smoothly with fewer errors from the TypeScript compiler to fix in each iteration. I can also search for variables declared as type `any` and find work that I still must do. "Any" tells the TypeScript compiler to not perform any type checking on that variable. It could be anything. The compiler will assume you know the APIs available on that variable. This doesn't mean every use of "any" is bad. There are valid uses for the `any` type, such as when a JavaScript API is designed to work with different types of objects. Using "any" as a placeholder during a migration is just one good form.

Finally, the declaration of `toggleShowDetails` shows how function declarations are represented in TypeScript:

```
toggleShowDetails: (contact: any) => boolean;
```

The function name is `toggleShowDetails`. After the colon, you'll see the parameter list. This function takes a single parameter, currently of type `any`. The name "contact" is optional. You can use this to provide more information to other programmers. The fat arrow points to the return type, which is a `boolean` in this example.

Having introduced the `any` type in the `IContactScope` definition shows you where to go to work next. TypeScript helps you avoid mistakes when you give it more information about the types with which you're working. I'll replace that `any` with a better definition of what's in a `Contact` by defining an `IContact` type that includes the properties available on a contact object (see **Figure 2**).

With the `IContact` interface now defined, I'll use it in the `IContactScope` interface:

```
interface IContactsScope extends ng.IScope {
    sortOrder: string;
    hideMessage: string;
    showMessage: string;
    contacts: IContact[];
    toggleShowDetails: (contact: IContact) => boolean;
}
```

I don't need to add type information on the definition of the `toggleShowDetails` function defined in the `contactsController` function. Because the `$scope` variable is an `IContactsScope`, the TypeScript compiler knows the function assigned to `toggleShowDetails` must match the function prototype defined in `IContactScope` and the parameter must be an `IContact`.

Look at the generated JavaScript for this version of the `contactsController` in **Figure 3**. Notice all the interface types I've defined have

Figure 3 The TypeScript Version of the Controller and the Generated JavaScript

```
/// reference path="../../Scripts/typings/angularjs/angular.d.ts"

var contactsApp: ng.IModule;

interface IContact {
    first: string;
    last: string;
    address: string;
    city: string;
    state: string;
    zipCode: number;
    cellPhone: number;
    homePhone: number;
    workPhone: number;
    showDetails: boolean
}

interface IContactsScope extends ng.IScope {
    sortOrder: string;
    hideMessage: string;
    showMessage: string;
    contacts: IContact[];
    toggleShowDetails: (contact: IContact) => boolean;
}

contactsApp.controller('ContactsController',
    function ContactsController($scope : IContactsScope, contactData) {
        $scope.sortOrder = 'last';
        $scope.hideMessage = "Hide Details";
        $scope.showMessage = "Show Details";
        $scope.contacts = contactData.getContacts();
        $scope.toggleShowDetails = function (contact) {
            contact.showDetails = !contact.showDetails;
            return contact.showDetails;
        }
    });

// Generated JavaScript
/// reference path="../../Scripts/typings/angularjs/angular.d.ts"
var contactsApp;

contactsApp.controller('ContactsController',
    function ContactsController($scope, contactData) {
        $scope.sortOrder = 'last';
        $scope.hideMessage = "Hide Details";
        $scope.showMessage = "Show Details";
        $scope.contacts = contactData.getContacts();
        $scope.toggleShowDetails = function (contact) {
            contact.showDetails = !contact.showDetails;
            return contact.showDetails;
        };
    });
///# sourceMappingURL=contactsController.js.map
```

PRECISELY PROGRAMMED FOR SPEED

DynamicPDF—Comprehensive PDF Solutions for .NET Developers

ceTe Software's DynamicPDF products provide real-time PDF generation, manipulation, conversion, printing, viewing, and much more. Providing the best of both worlds, the object models are extremely flexible but still supply the rich features you need as a developer. Reliable and efficient, the high-performance software is easy to learn and use. If you do encounter a question with any of our components, simply contact ceTe Software's readily available, industry-leading support team.



DynamicPDF

[WWW.DYNAMICPDF.COM](http://www.DynamicPDF.com)



TRY OUR PDF SOLUTIONS FREE TODAY!

www.DynamicPDF.com/eval or call 800.631.5006 | +1 410.772.8620

ceTe software

been removed from the generated JavaScript. The type annotations exist for you and for static analysis tools. These annotations don't carry through to the generated JavaScript because they aren't needed.

Add Module and Class Definitions

Adding type annotations to your code enables static analysis tools to find and report on possible mistakes you've made in your code. This encompasses everything from IntelliSense and lint-like analysis, to compile-time errors and warnings.

Another major advantage TypeScript provides over JavaScript is a better syntax to scope types. The TypeScript module keyword lets you place type definitions inside a scope and avoid collisions with types from other modules that may use the same name.

Adding type annotations to your code enables static analysis tools to find and report on possible mistakes you've made in your code.

The contacts sample application isn't that large, but it's still a good idea to place type definitions in modules to avoid collisions. Here, I'll place the `contactsController` and the other types I've defined inside a module named `Rolodex`:

```
module Rolodex {  
    // Elided  
}
```

I haven't added the `export` keyword on any definitions in this module. That means the types defined inside the `Rolodex` module can only be referenced from within that module. I'll add the `export` keyword on the interfaces defined in this module and use those types later as I migrate the `contactsData` code. I'll also change the code for the `ContactsController` from a function to a class. This class needs a constructor to initialize itself, but no other public methods (see **Figure 4**).

Creating this type now changes the call to `contactsApp.controller`. The second parameter is now the class type, not the function defined earlier. The first parameter of the controller function is the name of the controller. Angular maps controller names to constructor functions. Anywhere in the HTML page where the `ContactsController` type is referenced, Angular will call the constructor for the `ContactsController` class:

```
contactsApp.controller('ContactsController', Rolodex.ContactsController);
```

Figure 4 Change `ContactsController` from a Function to a Class

```
export class ContactsController {  
    constructor($scope: IContactsScope, contactData: any) {  
        $scope.sortOrder = 'last';  
        $scope.hideMessage = "Hide Details";  
        $scope.showMessage = "Show Details";  
        $scope.contacts = contactData.getContacts();  
        $scope.toggleShowDetails = function (contact) {  
            contact.showDetails = !contact.showDetails;  
            return contact.showDetails;  
        }  
    }  
}
```

The controller type has now been completely migrated from JavaScript to TypeScript. The new version contains type annotations for everything defined or used in the controller. In TypeScript, I could do that without needing changes in the other parts of the application. No other files were affected. Mixing TypeScript with JavaScript is smooth, which simplifies adding TypeScript to an existing JavaScript application. The TypeScript type system relies on type inference and structural typing, which facilitates easy interaction between TypeScript and JavaScript.

Now, I'll move on to the `contactData.js` file (see **Figure 5**). This function uses the Angular factory method to return an object that returns a list of contacts. Like the controller, the factory method maps names (`contactData`) to a function that returns the service. This convention is used in the controller's constructor. The second parameter of the constructor is named `contactData`. Angular uses that parameter name to map to the proper factory. As you can see, the Angular framework is convention-based.

Again, the first step is simply to change the extension from `.js` to `.ts`. It compiles cleanly and the generated JavaScript closely matches the source TypeScript file. Next, I'll put the code in the `contactData.ts`

Figure 5 The JavaScript Version of the `contactData` Service

```
'use strict';  
  
contactsApp.factory('contactData', function () {  
    var contacts = [  
        {  
            first: "Tom",  
            last: "Riddle",  
            address: "66 Shack St",  
            city: "Little Hangleton",  
            state: "Mississippi",  
            zipCode: 54565,  
            cellPhone: 6543654321,  
            homePhone: 4532332133,  
            workPhone: 6663420666  
        },  
        {  
            first: "Antonin",  
            last: "Dolohov",  
            address: "28 Kaban Ln",  
            city: "Gideon",  
            state: "Arkansas",  
            zipCode: 98767,  
            cellPhone: 4443332222,  
            homePhone: 5556667777,  
            workPhone: 9897876765  
        },  
        {  
            first: "Evan",  
            last: "Rosier",  
            address: "28 Dominion Ave",  
            city: "Notting",  
            state: "New Jersey",  
            zipCode: 23432,  
            cellPhone: 1232343456,  
            homePhone: 4432215565,  
            workPhone: 3454321234  
        }  
    ];  
  
    return {  
        getContacts: function () {  
            return contacts;  
        },  
        addContact: function (contact) {  
            contacts.push(contact);  
            return contacts;  
        }  
    };  
});
```

Creating a report is as easy as writing a letter



Reuse MS Word documents as your reporting templates



Create encrypted and print-ready Adobe PDF and PDF/A



Royalty-free WYSIWYG template designer



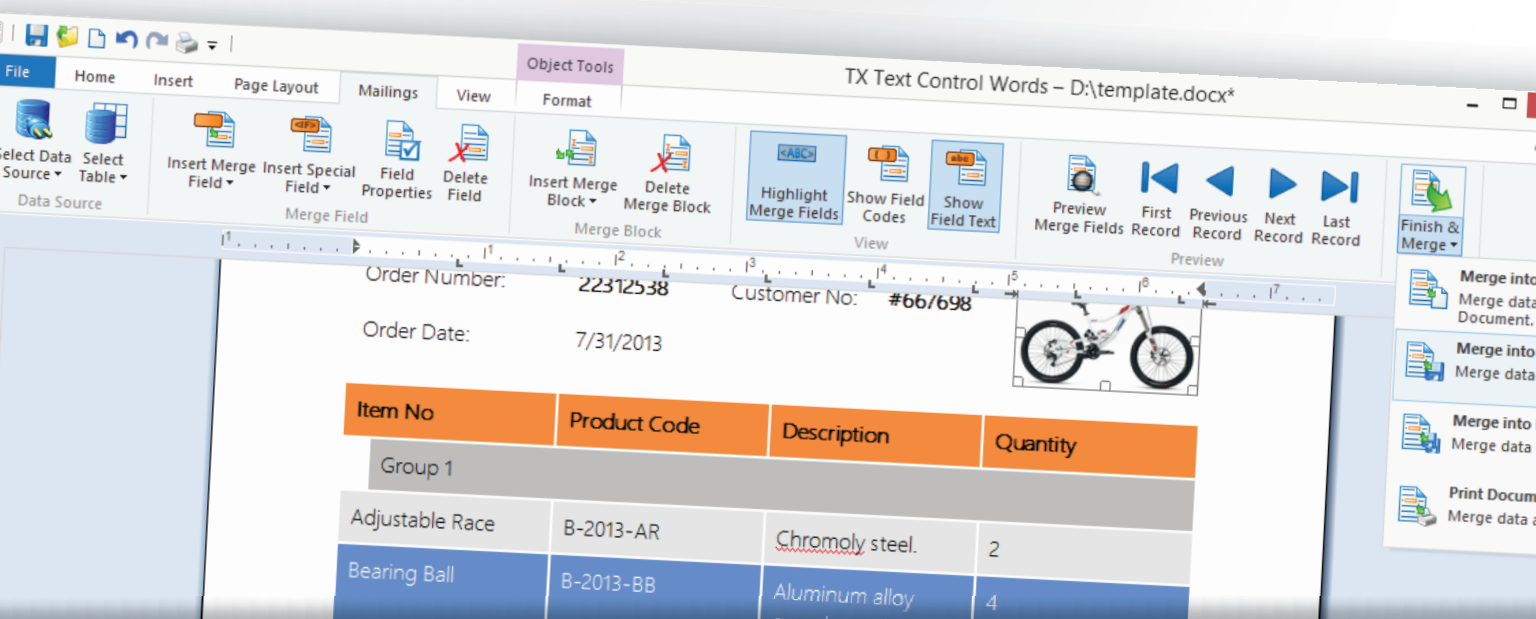
Powerful and dynamic 2D/3D charting support



Easy database connections and master-detail nested blocks



1D/2D barcode support including QRCode, IntelligentMail, EAN



www.textcontrol.com/reporting



txtextcontrol

US: +1 855-533-TEXT
EU: +49 421 427 067-10



Visual Studio

Microsoft

Partner

Reporting

Rich Text Editing

Spell Checking

Barcodes

PDF Reflow

Figure 6 The TypeScript Version of the ContactDataServer

```
/// reference path="../../Scripts/typings/angularjs/angular.d.ts"

var contactsApp: ng.IModule;

module Rolodex {
  export class ContactDataServer {
    contacts: IContact[] = [
      {
        first: "Tom",
        last: "Riddle",
        address: "66 Shack St",
        city: "Little Hangleton",
        state: "Mississippi",
        zipCode: 54565,
        cellPhone: 6543654321,
        homePhone: 4532332133,
        workPhone: 6663420666,
        showDetails: true
      },
      {
        first: "Antonin",
        last: "Dolohov",
        address: "28 Kaban Ln",
        city: "Gideon",
        state: "Arkansas",
        zipCode: 98767,
        cellPhone: 4443332222,
        homePhone: 5556667777,
        workPhone: 9897876765,
        showDetails: true
      },
      {
        first: "Evan",
        last: "Rosier",
        address: "28 Dominion Ave",
        city: "Notting",
        state: "New Jersey",
        zipCode: 23432,
        cellPhone: 1232343456,
        homePhone: 4432215565,
        workPhone: 3454321234,
        showDetails: true
      }
    ];

    getContacts() {
      return this.contacts;
    }

    addContact(contact: IContact) {
      this.contacts.push(contact);
      return this.contacts;
    }
  }
}

contactsApp.factory('contactsData', () => new Rolodex.ContactDataServer());
```

file in the same Rolodex module. That scopes all the code for the application in the same logical partition.

Next, I'll migrate the `contactData` factory to a class. Declare the class as the type `ContactDataServer`. Instead of a function that returns an object with two properties that are the methods, I can now simply define the methods as members of an object of `ContactDataServer`. The initial data is now a data member of an object of type `ContactDataServer`. I also need to use this type in the call to `contactsApp.factory`:

```
contactsApp.factory('contactsData', () => new Rolodex.ContactDataServer());
```

The second parameter is a function that returns a new `ContactDataServer`. The factory will create the object when I need it. If I try to compile and run this version, I'll have compiler errors because the `ContactDataServer` type isn't exported from the

Rolodex module. It is, however, referenced in the call to `contactsApp.factory`. This is another example of how the TypeScript type system is quite forgiving, which makes migration tasks much easier. I can easily fix this error by adding the `export` keyword to the `ContactDataServer` class declaration.

You can see the final version in **Figure 6**. Note that I've added type information for the array of contacts and the input parameter on the `addContact` method. The type annotations are optional—it's valid TypeScript without them. However, I'd encourage you to add all the necessary type information to your TypeScript code because it helps you avoid mistakes in the TypeScript system, which will allow you to be more productive.

Now that I've created a new `ContactDataServer` class, I can make one last change to the controller. Remember the `contactsController` constructor's second parameter was the data server. Now, I can make that more typesafe by declaring the following parameter must be of type `ContactDataServer`:

```
constructor($scope: IContactsScope, contactData: ContactDataServer) {
```

Smooth JavaScript to TypeScript Migrations

TypeScript has many more features than those I've demonstrated here. As you work with TypeScript, you'll embrace its capabilities. The more you use the TypeScript extensions to JavaScript, the more your productivity will increase. Remember that TypeScript type annotations were designed to provide a smooth migration from JavaScript to TypeScript. Most important, keep in mind TypeScript is a strict superset of JavaScript. That means any valid JavaScript is valid TypeScript.

Also, TypeScript type annotations have very little ceremony. Type annotations are checked where you provide them and you're not forced to add them everywhere. As you migrate from JavaScript to TypeScript, that's very helpful.

Finally, the TypeScript type system supports structural typing. As you define interfaces for important types, the TypeScript type system will assume any object with those methods and properties supports that interface. You don't need to declare interface support on each class definition. Anonymous objects can also support interfaces using this structural typing feature.

These features combined create a smooth path as you migrate your codebase from JavaScript to TypeScript. The further along the migration path you get, the more benefits you'll get from TypeScript static code analysis. Your end goal should be to leverage as much safety as possible from TypeScript. Along the way, your existing JavaScript code functions as valid TypeScript that doesn't make use of TypeScript's type annotations. It's an almost frictionless process. You don't have any reason not to use TypeScript in your current JavaScript applications. ■

BILL WAGNER is the author of the best-selling book, *"Effective C#" (2004)*, now in its second edition, and *"More Effective C#" (2008)*, both from Addison-Wesley Professional. He's also written videos for Pearson Education informIT, *"C# Async Fundamentals LiveLessons"* and *"C# Puzzlers."* He actively blogs at thebillwagner.com and can be reached at bill.w.wagner@outlook.com.

THANKS to the following Microsoft technical expert for reviewing this article:
Jonathan Turner



Extreme Performance Linear Scalability

For .NET & Java Apps

(Windows Azure & Amazon AWS Supported)

Cache data, reduce expensive database trips, and scale your apps to extreme transaction processing (XTP) with NCache.

In-Memory Distributed Cache

- Extremely fast & linearly scalable with 100% uptime
- Mirrored, Replicated, Partitioned, and Client Cache
- NHibernate & Entity Framework Level-2 Cache

ASP.NET Optimization in Web Farms

- ASP.NET Session State storage
- ASP.NET View State cache
- ASP.NET Output Cache provider
- ASP.NET JavaScript merge/minify

Runtime Data Sharing

- Powerful event notifications for pub/sub data sharing



Download a **FREE** trial!

sales@alachisoft.com

US: +1 (925) 236 3830

www.alachisoft.com

Override the Default Scaffold Templates

Jonathan Waldman

The mundane tasks of writing routine create, retrieve, update and delete operations against a data store is aptly conveyed by the oft-used acronym CRUD. Microsoft provides a helpful scaffolding engine, powered by T4 templates, that automates creating basic CRUD controllers and views for models in ASP.NET MVC applications that use the Entity Framework. (There are also WebAPI and MVC without Entity Framework scaffolders currently available.)

Scaffolds generate pages that are navigable and usable. They generally relieve you of the monotony involved in constructing CRUD pages. However, scaffolded results offer such limited functionality that you soon find yourself tweaking the generated controller logic and views to suit your needs.

The peril in doing this is that scaffolding is a one-way process. You can't re-scaffold your controller and views to reflect changes in a model without overwriting your tweaks. Therefore, you have to exercise vigilance in tracking which modules you've customized, so you can know which models you can safely re-scaffold and which ones you can't.

In a team environment, this vigilance is difficult to enforce. To top it off, the edit controller displays most model properties on the Edit view, thereby potentially exposing sensitive information. It blindly model binds and persists all view-submitted properties, which increases the risk of a mass-assignment attack.

In this article, I'll show you how to create project-specific customizations of the T4 templates that power the ASP.NET MVC Entity Framework CRUD scaffolding subsystem. Along the way, I'll show you how to extend the controller's Create and Edit post-back handlers so you can inject your own code between postback model binding and data-store persistence.

To address mass-assignment concerns, I'll create a custom attribute that gives you full control over the model properties that should persist and which ones shouldn't. Then I'll add another custom attribute that lets you display a property as a read-only label on the Edit view.

After this, you'll have unprecedented control over your CRUD pages and how your models are displayed and persisted while reducing your app's exposure to attacks. Best of all, you'll be able to leverage these techniques for all the models in your ASP.NET MVC projects, and safely regenerate controllers and views as your models change.

This article discusses:

- How to customize project-specific T4 templates
- Prevent mass-assignment security vulnerabilities
- Enforce a consistent look and feel to your CRUD pages

Technologies discussed:

Visual Studio 2013 Ultimate, ASP.NET MVC 5, Entity Framework 6, C#

Code download available at:

msdn.microsoft.com/magazine/msdnmag0614

Project Setup

I developed this solution using Visual Studio 2013 Ultimate, ASP.NET MVC 5, Entity Framework 6, and C# (the techniques discussed are also compatible with Visual Studio 2013 Professional, Premium and Express for Web and Visual Basic .NET). I created two solutions for download: The first is the Baseline Solution, which you can use to start with a working project and manually implement these techniques. The second is the Complete Solution, which includes all of the improvements discussed herein.

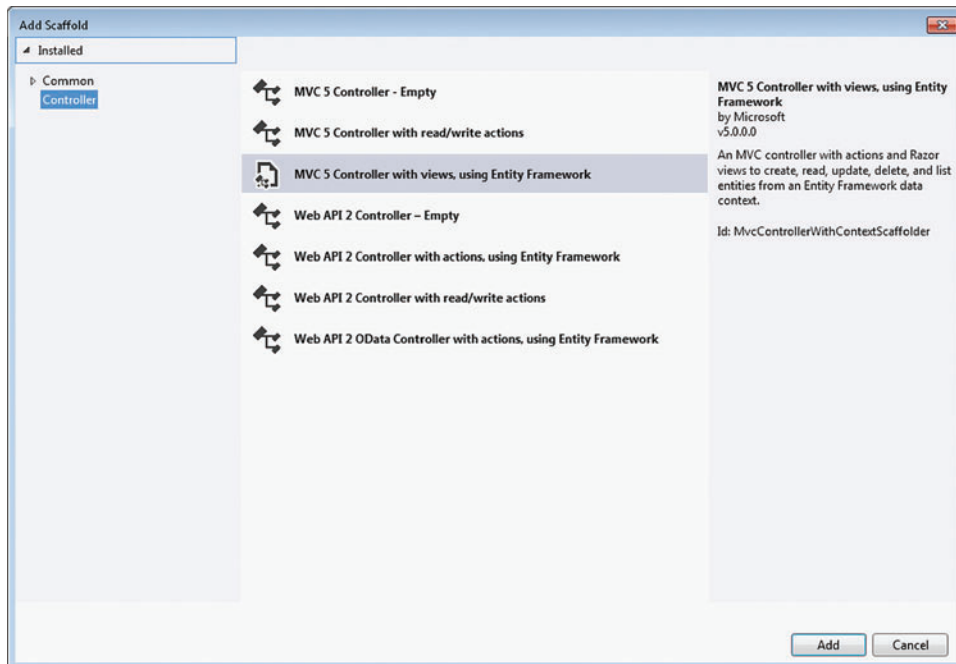


Figure 1 The MVC 5 Add Scaffold Dialog

Each solution contains three projects: one for the ASP.NET MVC Web site, one for entity models and T4-scaffold functions, and one for the data context. The solutions' data context points to a SQL Server Express database. In addition to the dependencies already mentioned, I added Bootstrap using NuGet to theme the scaffolded views.

The scaffold subsystem is installed during Setup when you check the Microsoft Web Developer Tools setup option. Subsequent Visual Studio service packs will update the scaffold files automatically. You can get any updates to the scaffold subsystem released between Visual Studio service packs in the latest Microsoft Web Platform Installer, which you can download from bit.ly/1g42AhP.

If you have any issues using the code download accompanying this article, ensure you have its latest version and carefully read the ReadMe.txt file. I'll update that as needed.

Define the Business Rules

To illustrate the full workflow involved in generating CRUD views and to reduce distractions, I'm going to work with a very simple entity model called Product.

```
public class Product
{
    public int ProductId { get; set; }
    public string Description { get; set; }
    public DateTime? CreatedDate { get; set; }
    public DateTime? ModifiedDate { get; set; }
}
```

By convention, MVC understands that ProductId is the primary key, but it has no idea I have special requirements regarding the CreatedDate and ModifiedDate properties. As their names imply, I want CreatedDate to convey when the product in question (represented by ProductId) was inserted into the database. I also want ModifiedDate to convey when it was last modified (I'll use UTC date-time values).

I'd like to display the ModifiedDate value on the Edit view as read-only text (if the record has never been modified, the ModifiedDate will equal the CreatedDate). I don't want to display the CreatedDate on any of the views. I also don't want the user to be able to supply or modify these date values, so I don't want to render any form controls that collect input for them on the Create or Edit views.

In an effort to make these values attack-proof, I want to be sure their values aren't persisted on postback, even if a clever hacker is able to supply them as form fields or querystring values. Because I consider these business logic layer rules, I don't want the database to have any responsibility for maintaining these column values (for instance, I don't want to create any

triggers or embed any table-column definition logic).

Explore the CRUD Scaffold Workflow

Let's first examine the default scaffold's function. I'll add a controller by right-clicking the Web project's Controllers folder and choosing Add Controller. This launches the Add Scaffold dialog (see Figure 1).

The entry "Mvc 5 Controller with views, using Entity Framework" is the one I'll use because it scaffolds the CRUD controller and views for a model. Select that entry and click Add. The next dialog gives you a number of options that end up as parameters for the T4 templates it subsequently transforms (see Figure 2).

Enter ProductController as the controller name. Keep "Use the async controller actions" checkbox unchecked for the sake of this discussion (async operations are beyond the scope of this article). Next, choose the Product model class. Because you're using the

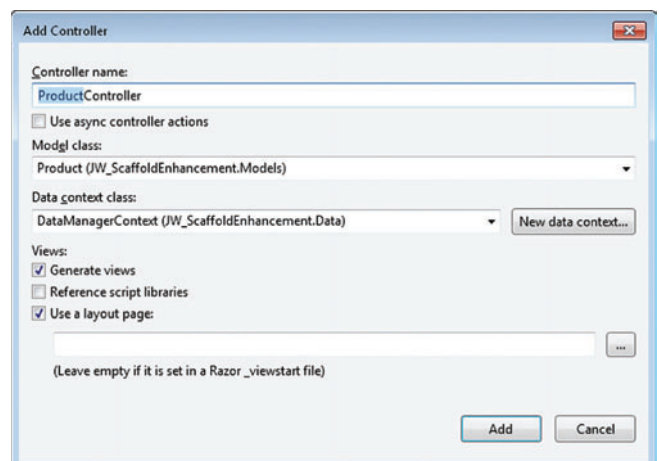


Figure 2 The Add Controller Dialog

Figure 3 The Scaffolded Controller Action Methods for Create and Edit

```
public ActionResult Create(
    [Bind(Include="ProductId,Description,CreateDate,ModifiedDate")] Product product)
{
    if (ModelState.IsValid)
    {
        db.Products.Add(product);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(product);
}

public ActionResult Edit(
    [Bind(Include="ProductId,Description,CreateDate,ModifiedDate")] Product product)
{
    if (ModelState.IsValid)
    {
        db.Entry(product).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(product);
}
```

Entity Framework, you'll need a data context class. Classes that inherit from System.Data.Entity.DbContext appear in the drop-down, so select the correct one if your solution uses more than one database context. With respect to the view options, check "Generate views" and "Use a layout page." Leave the layout page textbox empty.

When you click Add, several T4 templates are transformed to provide the scaffolded results. This process generates code for a controller (ProductController.cs), which is written to the Web project's Controllers folder, and five views (Create.cshtml, Delete.cshtml, Details.cshtml, Edit.cshtml, and Index.cshtml), which are written to the Web project's Views folder. At this point, you have a working controller and all the CRUD views you'll need to manage the data in the Product entity. You can begin using these Web pages right away, starting with the index view.

You'll likely want your CRUD pages to look and behave similarly for all the models in your project. Using T4 templates to scaffold the CRUD pages helps enforce this consistency. This means you should resist the temptation to directly modify controllers and views. Instead you should modify the T4 templates that generate

Figure 4 The T4 Templates That Scaffold Entity Framework CRUD Controller and Views

Scaffold Templates Sub-Folder Name	Template File Name (.cs for C#, .vb for Visual Basic .NET)	Generates This File (.cs for C#, .vb for Visual Basic .NET)
MvcControllerWithContext	Controller.cs.t4 Controller.vb.t4	Controller.cs Controller.vb
MvcView	Create.cs.t4 Create.vb.t4	Create.cshtml Create.vbhtml
MvcView	Delete.cs.t4 Delete.vb.t4	Delete.cshtml Delete.vbhtml
MvcView	Details.cs.t4 Details.vb.t4	Details.cshtml Details.vbhtml
MvcView	Edit.cs.t4 Edit.vb.t4	Edit.cshtml Edit.vbhtml
MvcView	Index.cshtml Index.vbhtml	Index.cshtml Index.vbhtml

them. Follow this practice to ensure your scaffolded files are ready to use with no further modifications.

Examine the Controller's Shortcomings

While the scaffold subsystem gets you up and running rather quickly, the controller it generates has several shortcomings. I'll show you how to make some improvements. Look at the scaffolded controller action methods that handle the Create and Edit in Figure 3.

The Bind attribute for each method explicitly includes every property on the Product model. When an MVC controller model binds all model properties after postback, it's called mass assignment. This is also called overposting and it's a serious security vulnerability. Hackers can exploit this vulnerability because there's a subsequent SaveChanges invocation on the database context. This ensures the model gets persisted to the data store. The controller template the CRUD scaffold system in MVC 5 uses generates mass-assignment code by default for the Create and Edit action postback methods.

You'll likely want your
CRUD pages to look and behave
similarly for all the models
in your project.

Another consequence of mass assignment happens if you choose to decorate certain properties on the model so they aren't rendered to the Create or Edit view. Those properties will be set to null after model binding. (See "Use Attributes to Suppress Properties on CRUD Views," which contains attributes you can use to specify whether you should render scaffolded properties to generated views.) To illustrate, I'll first add two attributes to the Product model:

```
public class Product
{
    public int ProductId { get; set; }
    public string Description { get; set; }
    [ScaffoldColumn(false)]
    public DateTime? CreatedDate { get; set; }
    [Editable(false)]
    public DateTime? ModifiedDate { get; set; }
}
```

When I re-run the scaffold process using Add Controller, as discussed earlier, the [Scaffold(false)] attribute ensures the CreatedDate won't appear on any views. The [Editable(false)] attribute ensures the ModifiedDate will appear on the Delete, Details, and Index views, but not the Create or Edit views. When properties aren't rendered to a Create or Edit view, they won't appear in the postback's HTTP request stream.

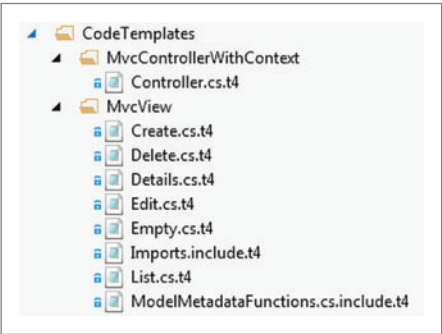


Figure 5 The Web Project's CodeTemplates

Spreadsheets Made Easy.



Fastest Calculations

Evaluate complex Excel-based models and business rules with the fastest and most complete Excel-compatible calculation engine available.



Comprehensive Charting

Enable users to visualize data with comprehensive Excel-compatible charting which makes creating, modifying, rendering and interacting with complex charts easier than ever before.



Windows
Forms



Silverlight



WPF

Powerful Controls

Add powerful Excel-compatible viewing, editing, formatting, calculating, filtering, sorting, charting, printing and more to your WinForms, WPF and Silverlight applications.



Scalable Reporting

Easily create richly formatted Excel reports without Excel from any ASP.NET, Windows Forms, WPF or Silverlight application.

Download your free fully functional evaluation at SpreadsheetGear.com



SpreadsheetGear

Toll Free USA (888) 774-3273 | Phone (913) 390-4797 | sales@spreadsheetgear.com

Figure 6 Extended Version of the Controller's Create Action Postback Method

```
public ActionResult Create(
    [Bind(Include="ProductId,Description,CreateDate,ModifiedDate")] Product product)
{
    if (ModelState.IsValid)
    {
        if (product is IControllerHooks) { ((IControllerHooks)product).OnCreate(); }
        db.Products.Add(product);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(product);
}
```

Figure 7 Extended Version of the Controller's Edit Action Postback Method

```
public ActionResult Edit(
    [Bind(Include="ProductId,Description,CreateDate,ModifiedDate")] Product product)
{
    if (ModelState.IsValid)
    {
        if (product is IControllerHooks) { ((IControllerHooks)product).OnEdit(); }
        db.Entry(product).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(product);
}
```

This is problematic because the last chance you have to assign values to model properties in these MVC-powered CRUD pages is during postback. So if a property postback value is null, that null value is going to be model-bound. Then the model is going to be persisted to the data store when `SaveChanges` is executed on the data context object. If this is done in the Edit postback action method, that property will be replaced with a null value. This effectively deletes the current value in the data store.

In my example, the value held by `CreateDate` in the data store would be lost. In fact, any property not rendered to the Edit view will cause the data store's value to be overwritten with null. If the model property or the data store doesn't allow assignment of a null value, you'll get an error on postback. To overcome these shortcomings, I'll modify the T4 template that's responsible for generating the controller.

Override the Scaffold Templates

To modify how the controller and views are scaffolded, you have to modify the T4 templates that generate them. You can modify the original templates, which will globally affect scaffolding across all Visual Studio projects. You could also modify project-specific copies of the T4 templates, which will only affect the project into which the copies are placed. I'll do the latter.

The original T4 scaffold templates are located in the `%programfiles%\Microsoft Visual Studio 12.0\Common7\IDE\Extensions\Microsoft\Web\Mvc\Scaffolding\Templates` folder. (These templates depend on several .NET assemblies, located in the `%programfiles%`

Figure 8 Product Model That Implements `IControllerHooks` to Extend the Controller

```
public class Product : IControllerHooks
{
    public int ProductId { get; set; }
    public string Description { get; set; }
    public DateTime? CreateDate { get; set; }
    public DateTime? ModifiedDate { get; set; }

    public void OnCreate()
    {
        this.CreateDate = DateTime.UtcNow;
        this.ModifiedDate = this.CreateDate;
    }

    public void OnEdit()
    {
        this.ModifiedDate = DateTime.UtcNow;
    }
}
```

Figure 9 The New Class Module `CustomAttributes.cs`

```
using System;
namespace JW_ScaffoldEnhancement.Models
{
    public class PersistPropertyOnEdit : Attribute
    {
        public readonly bool PersistPostbackDataFlag;
        public PersistPropertyOnEdit(bool persistPostbackDataFlag)
        {
            this.PersistPostbackDataFlag = persistPostbackDataFlag;
        }
    }
}
```

Microsoft Visual Studio 12.0\Common7\IDE\Extensions\Microsoft\Web Tools\Scaffolding folder.) I'll focus on the specific templates that scaffold the Entity Framework CRUD controller and views. These are summarized in Figure 4.

To create project-specific templates, copy the files you want to override from the original T4 scaffold folder to a folder in the ASP.NET MVC Web project called `CodeTemplates` (it must have this exact name). By convention, the scaffold subsystem first looks in the MVC project's `CodeTemplates` folder for a template match.

For this to work, you must precisely replicate the specific sub-folder names and file names you see in the original templates folder. I've copied the T4 files I plan to override from the CRUD for the Entity Framework scaffold subsystem. See my Web project's `CodeTemplates` in Figure 5.

I also copied `Imports.include.t4` and `ModelMetadataFunctions.cs.include.t4`. The project requires these files in order to scaffold the views. Also, I copied only the C# (.cs) versions of the files (if you're using Visual Basic .NET, you'll want to copy the files that include .vb in the file name). The scaffolding subsystem will transform these project-specific files, instead of their global versions.

```
<#
// get the Property IsModified source-code lines from our external assembly
List<string> list = ScaffoldFunctions.GetPropertyIsModifiedList("JW_ScaffoldEnhancement.Models", "Product");
foreach (string listItem in list)
{
    <#listItem#>
}
<#
#>
```

Figure 10 T4 Template Code Used To Generate an Improved Controller Edit Postback Handler

Extend the Create and Edit Action Methods

Now that I have project-specific T4 templates, I can modify them as needed. First, I'll extend the controller's Create and Edit action methods so I can inspect and modify the model before it's persisted. To keep the code the template generates as generic as possible, I don't want to add any model-specific logic to the template. Instead, I want to call an external function bound to the model. This way, the controller's Create and Edit are extended while they simulate polymorphism on the model. To this end, I'll create an interface and call it `IControllerHooks`:

```
namespace JW_ScaffoldEnhancement.Models
{
    public interface IControllerHooks
    {
        void OnCreate();
        void OnEdit();
    }
}
```

Using T4 templates to scaffold the CRUD pages helps enforce this consistency.

Next, I'll modify the `Controller.cs.t4` template (in the `CodeTemplates\MVCControllerWithContext` folder) so its Create and Edit postback action methods call the model's `OnCreate` and `OnEdit` methods,

Figure 11 The Model Project ScaffoldFunctions.GetPropertyIsModifiedList Static Function

```
static public List<string> GetPropertyIsModifiedList(string
ModelNamespace, string ModelTypeName,
string ModelVariable)
{
    List<string> OutputList = new List<string>();

    // Get the properties of the model object
    string aqn = Assembly.CreateQualifiedName(ModelNamespace + ", Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null", ModelNamespace + "." + ModelTypeName);

    // Get a Type object based on the Assembly Qualified Name
    Type typeModel = Type.GetType(aqn);
    // Get the properties of the type
    PropertyInfo[] typeModelProperties = typeModel.GetProperties();

    PersistPropertyOnEdit persistPropertyOnEdit;

    foreach (PropertyInfo propertyInfo in typeModelProperties)
    {
        persistPropertyOnEdit = (PersistPropertyOnEdit)Attribute.GetCustomAttribute(
            typeModel.GetProperty(propertyInfo.Name), typeof(PersistPropertyOnEdit));

        if (persistPropertyOnEdit == null)
        {
            OutputList.Add(ModelVariable + "Entry.Property(e => e." +
                propertyInfo.Name + ").IsModified = true;");
        }
        else
        {
            OutputList.Add(ModelVariable + "Entry.Property(e => e." +
                propertyInfo.Name + ").IsModified = " +
                ((PersistPropertyOnEdit)persistPropertyOnEdit).
                PersistPostbackDataFlag.ToString().ToLower() + ";");
        }
    }
    return OutputList;
}
```

Figure 12 The Newly Scaffolded Controller Edit Handler

```
if (ModelState.IsValid)
{
    if (product is IControllerHooks) { ((IControllerHooks)product).OnEdit(); }

    db.Products.Attach(product);
    var productEntry = db.Entry(product);

    productEntry.Property(e => e.ProductId).IsModified = true;
    productEntry.Property(e => e.Description).IsModified = true;
    productEntry.Property(e => e.CreatedDate).IsModified = false;
    productEntry.Property(e => e.ModifiedDate).IsModified = true;

    db.SaveChanges();
    return RedirectToAction("Index");
}
```

respectively, if the model has implemented `IControllerHooks`. The controller's Create action postback method is shown in **Figure 6** and its Create action postback method is shown in **Figure 7**.

Now, I'll modify the `Product` class so it implements `IControllerHooks`. Then I'll add the code I want to execute when the controller calls `OnCreate` and `OnEdit`. The new `Product` model class is shown in **Figure 8**.

Admittedly, there are many ways to implement this "extension" logic, but by using this one-line modification to the Create and Edit methods of the controller template, I can now modify the product model instance after model binding, but before persistence. I can even set the value of model properties that aren't published to the Create and Edit views.

You'll observe the model's `OnEdit` function doesn't set a value for `CreatedDate`. If `CreatedDate` isn't rendered to the Edit view, then it's going to be overwritten with a null value after the controller's Edit action method persists the model when it calls `SaveChanges`. To prevent this from happening, I'm going to need to make some further modifications to the controller template.

Figure 13 ScaffoldFunctions.GetHtmlForDisplayOnEditViewAttribute Static Function to Support the Custom DisplayOnEditViewFlag Attribute

```
static public string GetHtmlForDisplayOnEditViewAttribute(
    string ViewDataTypeName, string PropertyName, bool IsReadOnly)
{
    string returnValue = String.Empty;

    Attribute displayOnEditView = null;
    Type typeModel = Type.GetType(ViewDataTypeName);
    if (typeModel != null)
    {
        displayOnEditView =
            (DisplayOnEditView)Attribute.GetCustomAttribute(typeModel.GetProperty(
                PropertyName), typeof(DisplayOnEditView));
        if (displayOnEditView == null)
        {
            if (IsReadOnly)
            {
                returnValue = String.Empty;
            }
            else
            {
                returnValue = "@Html.EditorFor(model => model." + PropertyName + ")";
            }
        }
        else
        {
            if (((DisplayOnEditView)displayOnEditView).DisplayFlag == true)
            {
                returnValue = "@Html.DisplayTextFor(model => model." + PropertyName + ")";
            }
            else
            {
                returnValue = "@Html.EditorFor(model => model." + PropertyName + ")";
            }
        }
    }
    return returnValue;
}
```

Enhance the Edit Action Method

I've already mentioned some of the problems associated with mass assignment. One way to modify the model-binding behavior is to modify the Bind attribute so it excludes properties not to bind. In practice, however, this approach can still result in writing null values to the data store. A better strategy involves additional programming, but the payoff is well worth the effort.

I'm going to use the Entity Framework Attach method to attach the model to the database context. I can then track the entity entry and set the IsModified property as needed. To drive this logic, I'll create a new class module called CustomAttributes.cs in the JW_Scaffold-Enhancement.Models project (see Figure 9).

I'll use this attribute to indicate the properties I don't want to be persisted to the data store from the Edit view (undecorated properties will have an implicit [PersistPropertyOnEdit(true)] attribute). I'm interested in preventing the CreatedDate property from being persisted, so I've added the new attribute to only the CreatedDate property of my Product model. The newly decorated model class is shown here:

```
public class Product : IControllerHooks
{
    public int ProductId { get; set; }
    public string Description { get; set; }
    [PersistPropertyOnEdit(false)]
    public DateTime? CreatedDate { get; set; }
    public DateTime? ModifiedDate { get; set; }
}
```

Now I'll need to modify the Controller.cs.t4 template so it honors the new attribute. When enhancing a T4 template, you've got a choice as to whether to make changes within the template or external to the template. Unless you're using one of the third-party template-editor tools, I'd advocate placing as much code as possible in an external code module. Doing so provides a pure C# canvas (rather than one interspersed with T4 markup) within which you can focus on code. It also aids testing and gives you a way to incorporate your functions into your broader test-harness efforts. Finally, due to some shortcomings in how assemblies are referenced from a T4 scaffold, you'll experience fewer technical issues getting everything wired up.

My Models project contains a public function called GetPropertyIsModifiedList, which returns a List<String> I can iterate over to generate the IsModified settings for the assembly and type passed. Figure 10 shows this code in the Controller.cs.t4 template.



Figure 14 Edit View Showing a Read-Only ModifiedDate Field

In GetPropertyIsModifiedList, shown in Figure 11, I use reflection to gain access to the provided model's properties. Then I iterate across them to determine which ones are decorated with the PersistPropertyOnEdit attribute. You'll most likely want to persist most of the properties on your models, so I constructed the template code to set a property's IsModified value to true by default. This way, all you need to do is add [PersistPropertyOnEdit(false)] to the properties you don't want to persist.

The revised controller template generates a reconceived Edit postback action method, shown in Figure 12. My GetPropertyIsModifiedList function generates portions of this source code.

Customize a View

Sometimes you'll want to display a value on the edit view that you don't want users to edit. The ASP.NET MVC-provided attributes don't support this. I'd like to see the ModifiedDate on the Edit view, but I don't want the user to think it's an editable field. To implement this, I'll create another custom attribute called DisplayOnEditView in the CustomAttributes.cs class module, shown here:

```
public class DisplayOnEditView : Attribute
{
    public readonly bool DisplayFlag;
    public DisplayOnEditView(bool displayFlag)
    {
        this.DisplayFlag = displayFlag;
    }
}
```

This lets me decorate a model property so it renders as a label on the Edit view. Then I'll be able to show the ModifiedDate on the Edit view without worrying that someone can tamper with its value during postback.

Use Attributes to Suppress Properties on CRUD Views

ASP.NET MVC offers only three attributes that provide some control over whether a model's properties are rendered to the scaffolded views (see Figure A). The first two attributes do the same thing (although they reside in different namespaces): [Editable(false)] and [ReadOnly(true)]. These will cause the decorated property to not be rendered to the Create and Edit views. The third, [ScaffoldColumn(false)], causes the decorated property not to appear on any of the rendered views.

Figure A The Three Attributes That Prevent Properties from Rendering

Model Metadata Attribute	Attribute Namespace	Views Affected	What Happens
	None	None	No additional attributes causes only normal results.
[Editable(false)] [ReadOnly(true)]	Editable: System.ComponentModel.DataAnnotations ReadOnly: System.ComponentModel	Create Edit	Decorated model property not rendered.
[ScaffoldColumn(false)]	System.ComponentModel.DataAnnotations	Create Delete Details Edit Index	Decorated model property not rendered.

Now I can use that attribute to further decorate the Product model. I'll place the new attribute on the ModifiedDate property. I'll use [Editable(false)] to ensure it won't appear on the Create view and [DisplayOnEditView(true)] to ensure it appears as a label on the Edit view:

```
public class Product : IControllerHooks
{
    public int ProductId { get; set; }
    public string Description { get; set; }
    [PersistPropertyOnEdit(false)]
    [ScaffoldColumn(false)]
    public DateTime? CreatedDate { get; set; }
    [Editable(false)]
    [DisplayOnEditView(true)]
    public DateTime? ModifiedDate { get; set; }
}
```

Finally, I'll modify the T4 template that generates the Edit view so it honors the DisplayOnEditView attribute:

```
HtmlForDisplayOnEditViewAttribute =
    JW_ScaffoldEnhancement.Models.ScaffoldFunctions.
    GetHtmlForDisplayOnEditViewAttribute(
        ViewDataTypeName, property.PropertyName,
        property.IsReadOnly);
```

And I'll add the GetHtmlForDisplayOnEditViewAttribute function to the ScaffoldFunctions class as shown in Figure 13.

The GetHtmlForDisplayOnEditViewAttribute function returns Html.EditorFor when the attribute is false and Html.DisplayTextFor when the attribute is true. The Edit view will display the ModifiedDate as a label and all other non-key fields as editable textboxes, shown in Figure 14.

Wrapping Up

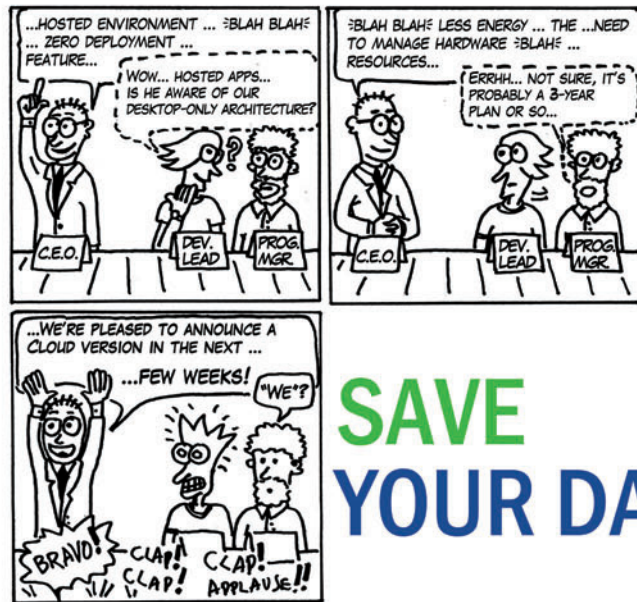
I've just scratched the surface of what you can accomplish with the scaffolding subsystem. I focused on the scaffolds that provide a CRUD control and views for the Entity Framework, but there are other scaffolds available to generate code for other types of Web pages and Web API actions.

If you've never worked with T4 templates, customizing existing templates is a great way to start. Although the templates I discussed here are launched from menus with the Visual Studio IDE, you can create custom T4 templates and transform them whenever necessary. Microsoft provides a good starting point at bit.ly/1coB616. If you're looking for something more advanced, I recommend Dustin Davis' course bit.ly/1bNIVXU.

At this time, Visual Studio 2013 doesn't include a robust T4 editor. In fact, it doesn't offer syntax highlighting or IntelliSense. Fortunately, there are some add-on products that do. Check out the Devart T4 editor (bit.ly/1cabz0E) and the Tangible Engineering T4 Editor (bit.ly/1fswFbo). ■

JONATHAN WALDMAN is an experienced software developer and technical architect. He has been working with the Microsoft technology stack since its inception. He has worked on several high-profile enterprise projects and participated in all aspects of the software development lifecycle. He's a member of the Pluralsight technical team and continues to develop software solutions and educational materials. Reach him at jonathan.waldman@live.com.

THANKS to the following Microsoft technical expert for reviewing this article:
Joost de Nijs



**SAVE
YOUR DAY!**



**Use
CodeFluent
Entities**

CodeFluent Entities is a unique product integrated into Visual Studio that allows you to generate database scripts, code (C#, VB), web services and UIs.

"I recently spent a week attending a course on Entity Framework but CodeFluent Entities provides so much more and is decidedly easier to understand and implement" *

Peter Stanford - Artefaction - Australia

* Source : <http://visualstudiogallery.msdn.microsoft.com/86299BBF-1EF1-436D-B618-66E8C16AB410>

To get a license worth \$399 for free
Go to www.softfluent.com/forms/msdn-2014

CodeFluent Entities
tools for developers, by developers

More information: www.softfluent.com Contact us: info@softfluent.com

Visual Studio **LIVE!**
EXPERT SOLUTIONS FOR .NET DEVELOPERS

MICROSOFT HEADQUARTERS
REDMOND 2014



SET YOUR COURSE FOR 127.0.0.1!

vslive.com/redmond

EVENT SPONSOR
Microsoft

PLATINUM SPONSOR
 **esri**

SPONSOR
 **LogiGear**
Software Testing

SUPPORTED BY
 **Visual Studio**

msdn
magazine

Visual Studio
MAGAZINE

REDMOND 2014

August 18 – 22
Microsoft Headquarters
Redmond, WA



YOUR GUIDE TO THE .NET DEVELOPMENT UNIVERSE

From August 18 – 22, 2014, developers, engineers, software architects and designers will land in Redmond, WA at the idyllic Microsoft headquarters for 5 days of cutting-edge education on the Microsoft Platform.

Come experience code at the source – rub elbows with Microsoft stars, get the inside scoop on what's next, and learn what you need to know now. Over 5 days and 60+ sessions and workshops, you'll explore the .NET Development Universe, receiving the kind of practical, unbiased training you can only get at Visual Studio Live!

Tracks Include:

- Visual Studio/.NET Framework
- Windows Client
- JavaScript/HTML5 Client
- ASP.NET
- Cloud Computing
- Windows Phone
- Cross-Platform Mobile Development
- SharePoint
- SQL Server



**Register by June 11
and Save \$400!**

Use Promo Code REDJUN4



Scan the QR code to
register or for more
event details.

vslive.com/redmond

PRODUCED BY



TURN THE PAGE FOR
MORE EVENT DETAILS

Visual Studio Live! has partnered with the Hyatt Regency Bellevue for conference attendees at a special reduced rate.



AGENDA AT-A-GLANCE

Visual Studio / .NET Framework	Windows Client
--------------------------------	----------------

START TIME	END TIME	
7:00 AM	8:00 AM	
8:00 AM	12:00 PM	MW01 - Workshop: Modern UX Design - Billy Hollis
12:00 PM	2:30 PM	
2:30 PM	6:00 PM	MW01 - Workshop: Modern UX Design - Billy Hollis
7:00 PM	9:00 PM	

START TIME	END TIME	
7:30 AM	8:30 AM	
8:30 AM	9:30 AM	
9:45 AM	11:00 AM	T01 - What's New in WinRT Development - Rockford Lhotka
11:15 AM	12:30 PM	T06 - What's New for XAML Windows Store Apps - Ben Dewey
12:30 PM	2:30 PM	
2:30 PM	3:45 PM	T11 - Interaction Design Principles and Patterns - Billy Hollis
3:45 PM	4:15 PM	
4:15 PM	5:30 PM	T16 - Applying UX Design in XAML - Billy Hollis
5:30 PM	7:00 PM	

START TIME	END TIME	
7:30 AM	8:00 AM	
8:00 AM	9:00 AM	
9:15 AM	10:30 AM	W01 - Getting Started with Windows Phone Development - Nick Landry
10:45 AM	12:00 PM	W06 - Build Your First Mobile App in 1 hour with Microsoft App Studio - Nick Landry
12:00 PM	1:30 PM	
1:30 PM	2:45 PM	W11 - What's New for HTML/WinJS Windows Store Apps - Ben Dewey
2:45 PM	3:15 PM	
3:15 PM	4:30 PM	W16 - Windows 8 HTML/JS Apps for the ASP.NET Developer - Adam Tuliper
8:00 PM	10:00 PM	

START TIME	END TIME	
7:30 AM	8:00 AM	
8:00 AM	9:15 AM	TH01 - Developing Awesome 3D Applications with Unity and C#/JavaScript - Adam Tuliper
9:30 AM	10:45 AM	TH06 - What's New in WPF 4.5 - Walt Ritscher
11:00 AM	12:15 PM	TH11 - Implementing M-V-VM (Model-View-View Model) for WPF - Philip Japikse
12:15 PM	2:15 PM	
2:15 PM	3:30 PM	TH16 - Build Maintainable Windows Store Apps with MVVM and Prism - Brian Noyes
3:45 PM	5:00 PM	TH21 - Make Your App Alive with Tiles and Notifications - Ben Dewey

START TIME	END TIME	
7:30 AM	8:00 AM	
8:00 AM	12:00 PM	FW01 - Workshop: Service Orientation
12:00 PM	1:00 PM	
1:00 PM	5:00 PM	FW01 - Workshop: Service Orientation

Speakers and sessions subject to change

CONNECT WITH VISUAL STUDIO LIVE!



twitter.com/vslive - @VSLive



facebook.com - Search "VSLive"



linkedin.com - Join the "Visual Studio Live" group!



Register at
vslive.com/redmond
Use Promo Code REDJUN4

Scan the QR code to register or for more event details.

August 18 – 22 | Microsoft Headquarters | Redmond, WA

Cloud Computing	Windows Phone	Cross-Platform Mobile Development	ASP.NET	JavaScript / HTML5 Client	SharePoint	SQL Server
-----------------	---------------	-----------------------------------	---------	---------------------------	------------	------------

Visual Studio Live! Pre-Conference Workshops: Monday, August 18, 2014 (Separate entry fee required)

Pre-Conference Workshop Registration - Coffee and Morning Pastries

MW02 - Workshop: Data-Centric Single Page Applications with Durandal, Knockout, Breeze, and Web API - Brian Noyes

MW03 - Workshop: SQL Server for Developers - Andrew Brust & Leonard Label

Lunch @ The Mixer - Visit the Microsoft Company Store & Visitor Center

MW02 - Workshop: Data-Centric Single Page Applications with Durandal, Knockout, Breeze, and Web API - Brian Noyes

MW03 - Workshop: SQL Server for Developers - Andrew Brust & Leonard Label

Dine-A-Round Dinner

Visual Studio Live! Day 1: Tuesday, August 19, 2014

Registration - Coffee and Morning Pastries

Keynote: To Be Announced

T02 - Creating Data-Driven Mobile Web Apps with ASP.NET MVC and jQuery Mobile - Rachel Appel

T03 - HTML5 for Better Web Sites - Robert Boedigheimer

T04 - Introduction to Windows Azure - Vishwas Lele

T05 - What's New in the Visual Studio 2013 IDE

T07 - To Be Announced

T08 - Great User Experiences with CSS 3 - Robert Boedigheimer

T09 - Windows Azure Cloud Services - Vishwas Lele

T10 - What's New for Web Developers in Visual Studio this Year? - Mads Kristensen

Lunch - Visit Exhibitors

T12 - Getting Started with Xamarin - Walt Ritscher

T13 - Building Real Time Applications with ASP.NET SignalR - Rachel Appel

T14 - To Be Announced

T15 - Why Browser Link Changes Things, and How You Can Write Extensions? - Mads Kristensen

Sponsored Break - Visit Exhibitors

T17 - Building Multi-Platform Mobile Apps with Push Notifications - Nick Landry

T18 - JavaScript for the C# Developer - Philip Japikse

T19 - Windows Azure SQL Database - SQL Server in the Cloud - Leonard Label

T20 - Katana, OWIN, and Other Awesome Codenames: What's coming? - Howard Dierking

Microsoft Ask the Experts & Exhibitor Reception - Attend Exhibitor Demos

Visual Studio Live! Day 2: Wednesday, August 20, 2014

Registration - Coffee and Morning Pastries

Keynote: To Be Announced

W02 - Learning Entity Framework 6 - Leonard Label

W03 - Build Data-Centric HTML5 Single Page Applications with Breeze - Brian Noyes

W04 - To Be Announced

W05 - Upgrading Your Existing ASP.NET Apps - Pranav Rastogi

W07 - Programming the T-SQL Enhancements in SQL Server 2012 - Leonard Label

W08 - Knocking It Out of the Park, with KnockoutJS - Miguel Castro

W09 - Solving Security and Compliance Challenges with Hybrid Clouds - Eric D. Boyd

W10 - Creating Map Centric Applications for Windows, WinRT and Window Phone - Ben Ramseth

Birds-of-a-Feather Lunch - Visit Exhibitors

W12 - SQL Server 2014: Features Drill-down - Scott Klein

W13 - JavaScript: Turtles, All the Way Down - Ted Neward

W14 - Zero to Connected with Windows Azure Mobile Services - Brian Noyes

W15 - To Be Announced

Sponsored Break - Exhibitor Raffle @ 2:55 pm (Must be present to win)

W17 - SQL Server 2014 In-memory OLTP - Deep Dive - Scott Klein

W18 - AngularJS JumpStart - Brian Noyes

W19 - Leveraging Azure Web Sites - Rockford Lhotka

W20 - Cool ALM Features in Visual Studio 2013 - Brian Randell

Lucky Strike Evening Out Party

Visual Studio Live! Day 3: Thursday, August 21, 2014

Registration - Coffee and Morning Pastries

TH02 - AWS for the SQL Server Professional - Lynn Langit

TH03 - Sexy Extensibility Patterns - Miguel Castro

TH04 - Beyond Hello World: A Practical Introduction to Node.js on Windows Azure Websites - Rick Garibay

TH05 - Leveraging Visual Studio Online - Brian Randell

TH07 - Real-world Predictive Analytics with PowerBI and Predixion Software - Lynn Langit

TH08 - What's New in MVC 5 - Miguel Castro

TH09 - From the Internet of Things to Intelligent Systems: A Developer's Primer - Rick Garibay

TH10 - Essential C# 6.0 - Mark Michaelis

TH12 - Excel, Power BI and You: An Analytics Superhub - Andrew Brust

TH13 - What's New in Web API 2 - Miguel Castro

TH14 - Building Mobile Applications with SharePoint 2013 - Darrin Bishop

TH15 - Performance and Diagnostics Hub in Visual Studio 2013 - Brian Peek

Lunch @ The Mixer - Visit the Microsoft Company Store & Visitor Center

TH17 - Big Data 101 with HDInsight - Andrew Brust

TH18 - Finding and Consuming Public Data APIs - G. Andrew Duthie

TH19 - Building Apps for SharePoint - Mark Michaelis

TH20 - Git for the Microsoft Developer - Eric D. Boyd

TH22 - NoSQL for the SQL Guy - Ted Neward

TH23 - Provide Value to Customers and Enhance Site Stickiness By Creating an API - G. Andrew Duthie

TH24 - Data as Information: Understanding Your SharePoint 2013 Business Intelligence Options - Darrin Bishop

TH25 - Writing Asynchronous Code Using .NET 4.5 and C# 5.0 - Brian Peek

Visual Studio Live! Post-Conference Workshops: Friday, August 22, 2014 (Separate entry fee required)

Post-Conference Workshop Registration - Coffee and Morning Pastries

Technologies: Designing, Developing, & Implementing WCF and the Web API - Miguel Castro

FW02 - Workshop: A Day of Windows Azure - Eric D. Boyd

Lunch

Technologies: Designing, Developing, & Implementing WCF and the Web API - Miguel Castro

FW02 - Workshop: A Day of Windows Azure - Eric D. Boyd

vslive.com/redmond

Topshelf and Katana: A Unified Web and Service Architecture

Wes McClure

Using IIS to host ASP.NET Web applications has been the de facto standard for more than a decade. Building such applications is a relatively simple process, but deploying them is not. Deploying requires curated knowledge of application configuration hierarchies and nuances of the history of IIS, and the tedious provisioning of sites, applications and virtual directories. Many of the critical pieces of infrastructure often end up living outside the application in manually configured IIS components.

When applications outgrow simple Web requests and need to support long-running requests, recurring jobs and other processing work, they become very difficult to support within IIS. Often, the solution is to create a separate Windows Service to host these components. But this requires an entirely separate deployment process, doubling the effort involved. The last straw is getting the Web and service processes to communicate. What could be a very simple application quickly becomes extremely complex.

This article discusses:

- Traditional separated Web and service architecture
- Unifying Web and service components with Katana and Topshelf
- Building a simple SMS messaging demo application

Technologies discussed:

ASP.NET, Katana Project, OWIN, Topshelf, Nancy

Code download available at:

msdn.microsoft.com/magazine/msdnmag0614

Figure 1 shows what this architecture typically looks like. The Web layer is responsible for handling fast requests and providing a UI to the system. Long-running requests are delegated to the service, which also handles recurring jobs and processing. In addition, the service provides status about current and future work to the Web layer to be included in the UI.

A New Approach

Fortunately, new technologies are emerging that can make developing and deploying Web and service applications much simpler. Thanks to the Katana project (katanaproject.codeplex.com) and the specifications provided by OWIN (owin.org), it's now possible to self-host Web applications, taking IIS out of the equation, and still support many of the ubiquitous ASP.NET components such as WebApi and SignalR. The Web self-host can be embedded in a rudimentary console application along with Topshelf (topshelf-project.com) to create a Windows service with ease. As a result, Web and service components can live side-by-side in the same process, as shown in Figure 2. This eliminates the overhead of developing extraneous communication layers, separate projects and separate deployment procedures.

This capability isn't entirely new. Topshelf has been around for years, helping to simplify Windows Service development, and there are many open source self-host Web frameworks, such as Nancy. However, until OWIN blossomed into the Katana project, nothing has shown as much promise of becoming the de facto standard alternative for hosting Web applications in IIS. Moreover, Nancy and many open source components work with the Katana project, allowing you to curate an eclectic, supple framework.

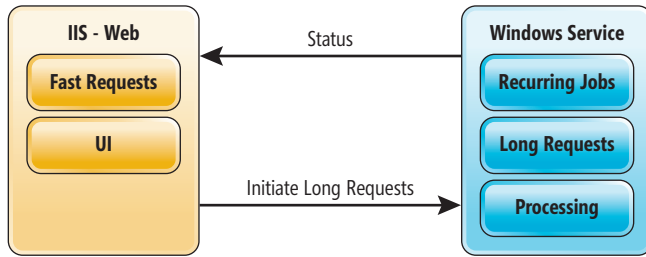


Figure 1 Traditional Separated Web and Service Architecture

Topshelf may seem optional, but that's not the case. Without the ability to simplify service development, self-hosted Web development can be extremely cumbersome. Topshelf simplifies developing the service by treating it as a console application and abstracting the fact that it will be hosted as a service. When it's time to deploy, Topshelf automatically handles installing and starting the application as a Windows Service—all without the overhead of dealing with InstallUtil; the nuances of a service project and service components; and attaching debuggers to services when something goes wrong. Topshelf also allows many parameters, such as the service name, to be specified in code or configured during installation via the command line.

When applications outgrow Web requests and need to support long-running requests, recurring jobs and other processing work, they become very difficult to support within IIS.

To illustrate how to unify Web and service components with Katana and Topshelf, I'll build a simple SMS messaging application. I'll start out with an API to receive messages and queue them for sending. This will demonstrate how easy it is to handle long-running requests. Then, I'll add an API query method to return the count of pending messages, showing it's also easy to query service status from Web components.

Next, I'll add an administrative interface to demonstrate self-hosted Web components still afford building rich Web interfaces. To round out the message processing, I'll add a component to send messages as they're queued, to showcase including service components.

And to highlight one of the best parts of this architecture, I'll create a psake script to expose the simplicity of deployments.

To focus on the combined benefits of Katana and Topshelf, I won't go into detail about either project. Check out "Getting

Started with the Katana Project" (bit.ly/1h9XaBL) and "Create Windows Services Easily with Topshelf" (bit.ly/1h9XReh) to learn more.

A Console Application Is All You Need To Get Started

Topshelf exists to easily develop and deploy a Windows service from the starting point of a simple console application. To get started with the SMS messaging application, I create a C# console application and then install the Topshelf NuGet package from the Package Manager Console.

When the console application starts, I need to configure the Topshelf HostFactory to abstract hosting the application as a console in development and as a service in production:

```
private static int Main()
{
    var exitCode = HostFactory.Run(host =>
    {
    });
    return (int) exitCode;
}
```

The HostFactory will return an exit code, which is helpful during service installation to detect and stop on failure. The host configurator provides a service method to specify a custom type that represents the entry point to your application code. Topshelf refers to this as the service it's hosting, because Topshelf is a framework for simplifying Windows Service creation:

```
host.Service<SmsApplication>(service =>
{
});
```

Next, I create an SmsApplication type to contain logic to spin up the self-hosted Web server and the traditional Windows service components. At a minimum, this custom type will contain behavior to execute when the application starts or stops:

```
public class SmsApplication
{
    public void Start()
    {
    }

    public void Stop()
    {
    }
}
```

Because I chose to use a plain old CLR object (POCO) for the service type, I provide a lambda expression to Topshelf to construct an instance of the SmsApplication type, and I specify the start and stop methods:

```
service.ConstructUsing(() => new SmsApplication());
service.WhenStarted(a => a.Start());
service.WhenStopped(a => a.Stop());
```

Topshelf allows many service parameters to be configured in code, so I use SetDescription, SetDisplayName and SetServiceName to describe and name the service that will be installed in production:

```
host.SetDescription("An application to manage sending sms messages and provide message status.");
host.SetDisplayName("Sms Messaging");
host.SetServiceName("SmsMessaging");
host.RunAsNetworkService();
```

Finally, I use RunAsNetworkService to instruct Topshelf to configure the service to run as the network service account. You can

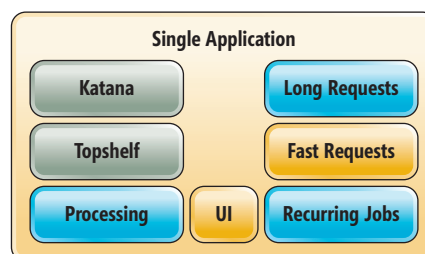


Figure 2 Unified Web and Service Architecture with Katana and Topshelf

change this to whatever account fits your environment. For more service options, see the Topshelf configuration documentation at bit.ly/1AfMIQ.

Running the service as a console application is as simple as launching the executable. **Figure 3** shows the output of starting and stopping the SMS executable. Because this is a console application, when you launch your application in Visual Studio, you'll experience the same behavior.

Incorporating an API

With the Topshelf plumbing in place I can begin work on the application's API. The Katana project provides components to self-host an OWIN pipeline, so I install the Microsoft.Owin.SelfHost package to incorporate the self-host components. This package references several packages, two of which are important for self-hosting. First, Microsoft.Owin.Hosting provides a set of components to host and run an OWIN pipeline. Second, Microsoft.Owin.Host.HttpListener supplies an implementation of an HTTP server.

Nancy and many
open source components
work with the Katana project,
allowing you to curate an
eclectic, supple framework.

Inside the `SmsApplication`, I create a self-hosted Web application using the `WebApp` type provided by the Hosting package:

```
protected IDisposable WebApplication;

public void Start()
{
    WebApplication = WebApp.Start<WebPipeline>("http://localhost:5000");
}
```

The `WebApp` `Start` method requires two parameters, a generic parameter to specify a type that will configure the OWIN pipeline, and a URL to listen for requests. The Web application is a disposable resource. When the `SmsApplication` instance is stopped, I dispose of the Web application:

```
public void Stop()
{
    WebApplication.Dispose();
}
```

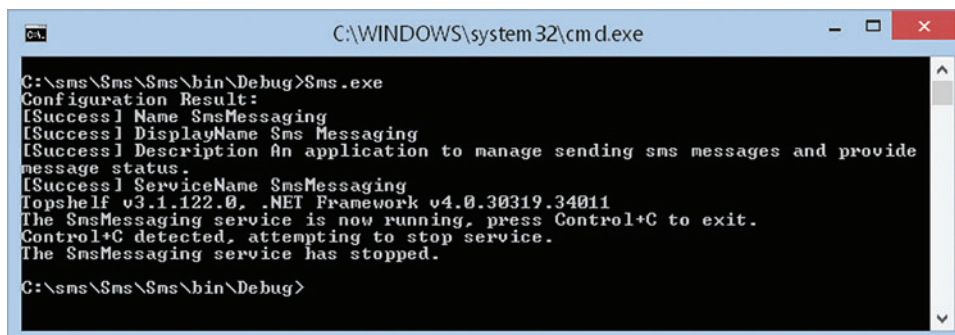


Figure 3 Running the Service as a Console Application

One benefit of using OWIN is that I can leverage a variety of familiar components. First, I'll use `WebApi` to create the API. I need to install the `Microsoft.AspNet.WebApi.Owin` package to incorporate `WebApi` in the OWIN pipeline. Then, I'll create the `WebPipeline` type to configure the OWIN pipeline and inject the `WebApi` middleware. Additionally, I'll configure `WebApi` to use attribute routing:

```
public class WebPipeline
{
    public void Configuration(IApplicationBuilder application)
    {
        var config = new HttpConfiguration();
        config.MapHttpAttributeRoutes();
        application.UseWebApi(config);
    }
}
```

And now I can create an API method to receive messages and queue them to be sent:

```
public class MessageController : ApiController
{
    [Route("api/messages/send")]
    public void Send([FromUri] SmsMessageDetails message)
    {
        MessageQueue.Messages.Add(message);
    }
}
```

`SmsMessageDetails` contains the payload of the message. The `send` action adds the message to a queue to be asynchronously processed at a later time. The `MessageQueue` is a global `BlockingCollection`. In a real application this might mean you need to factor in other concerns such as durability and scalability:

```
public static readonly BlockingCollection<SmsMessageDetails> Messages;
```

In a separated Web and service architecture, handing off asynchronous processing of long-running requests, such as sending a message, requires communication between the Web and service processes. And adding API methods to query the status of the service means even more communication overhead. A unified approach makes sharing status information between Web and service components simple. To demonstrate this, I add a `PendingCount` query to the API:

```
[Route("api/messages/pending")]
public int PendingCount()
{
    return MessageQueue.Messages.Count;
}
```

Building a Rich UI

APIs are convenient, but self-hosted Web applications still need to support a visual interface. In the future, I suspect, the ASP.NET MVC framework or a derivative will be available as OWIN middleware. For now, Nancy is compatible and has a package to support the core of the Razor view engine.

I'll install the `Nancy.Owin` package to add support for Nancy, and the `Nancy.Viewengines.Razor` to incorporate the Razor view engine. To plug Nancy into the OWIN pipeline, I need to register it after the registration for `WebApi` so it doesn't capture the routes I mapped

GdPicture.NET 10



- Document viewing, processing, printing and scanning (TWAIN & WIA).
- Reading, writing and converting vector and raster images in more than 90 formats, PDF included.
- OMR, OCR, barcode reading and writing (linear & 2D).
- Annotations for image and PDF within Windows and Web applications.
- Color detection engine for image and PDF compression.

And much more ... 

All-In-One Document Imaging SDK

Royalty-Free Document Imaging Toolkits
for .NET and COM/ActiveX



GdPicture.NET 10 Plugins



Color detection



DICOM image reader



MICR reader

- Full managed PDF support
- Full annotations support for PDF and images
- OCR
- Forms processing
- JBIG2 encoding
- 1D and 2D barcode reading and writing



Try GdPicture.NET 10
FREE for 30 days

www.gdpicture.com

to the API. By default, Nancy returns an error if a resource isn't found, whereas WebApi passes requests it can't handle back to the pipeline:

```
application.UseNancy();
```

To learn more about using Nancy with an OWIN pipeline, refer to “Hosting Nancy with OWIN” at bit.ly/1gqjlly.

To build an administrative status interface, I add a Nancy module and map a status route to render a status view, passing the pending message count as the view model:

```
public class StatusModule : NancyModule
{
    public StatusModule()
    {
        Get["/status"] =
            _ => View["status", MessageQueue.Messages.Count];
    }
}
```

The view isn't very glamorous at this point, just a simple count of pending messages:

```
<h2>Status</h2>
There are <strong>@Model</strong> messages pending.
```

I'm going to jazz up the view a bit with a simple Bootstrap navbar, as shown in **Figure 4**. Using Bootstrap requires hosting static content for the Bootstrap style sheet.

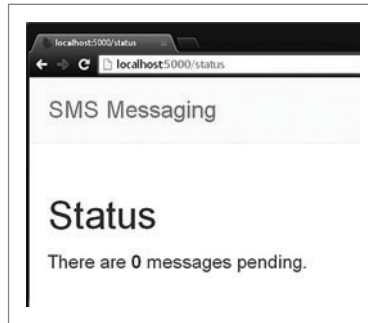


Figure 4 Administrative Status Page

Without the ability to simplify
service development,
self-hosted Web development
can be quite cumbersome.

I could use Nancy to host static content, but the advantage of OWIN is mixing and matching middleware, so instead I'm going to use the newly released Microsoft.Owin.StaticFiles package, which is part of the Katana project. The StaticFiles package provides file-serving middleware. I'll add it to the start of the OWIN pipeline so the Nancy static file serving doesn't kick in.

```
application.UseFileServer(new FileServerOptions
{
    FileSystem = new PhysicalFileSystem("static"),
    RequestPath = new PathString("/static")
});
```

The FileSystem parameter tells the file server where to look for files to serve. I'm using a folder named static. The RequestPath specifies the route prefix to listen for requests for this content. In this case, I chose to mirror the name static, but these don't have to match. I use the following link in the layout to reference the bootstrap style sheet (naturally, I place the Bootstrap style sheet in a CSS folder inside the static folder):

```
<link rel="stylesheet" href="/static/css/bootstrap.min.css">
```

A Word About Static Content and Views

Before I move on, I want to tell you about a tip I find helpful when developing a self-hosted Web application. Normally, you'd set the static content and MVC views to be copied to the output directory

so the self-hosted Web components can find them relative to the current executing assembly. Not only is this a burden and easy to forget, changing the views and static content requires recompiling the application—which absolutely kills productivity. Therefore, I recommend not copying the static content and views to the output directory; instead, configure middleware such as Nancy and the FileServer to map to the development folders.

By default, the debug output directory of a console application is bin/Debug, so in

development I tell the FileServer to look two directories above the current directory to find the static folder that contains the Bootstrap style sheet:

```
FileSystem = new PhysicalFileSystem(IsDevelopment() ? "../..static" : "static")
```

Then, to tell Nancy where to look for views, I'll create a custom NancyPathProvider:

```
public class NancyPathProvider : IRootPathProvider
{
    public string GetRootPath()
    {
        return WebPipeline.IsDevelopment()
            ? Path.Combine(AppDomain.CurrentDomain.BaseDirectory, @"..\..\")
            : AppDomain.CurrentDomain.BaseDirectory;
    }
}
```

Again, I use the same check to look two directories above the base directory if I'm running in development mode in Visual Studio. I've left the implementation of IsDevelopment up to you; it could be a simple configuration setting or you could write code to detect when the application was launched from Visual Studio.

To register this custom root path provider, I create a custom NancyBootstrapper and override the default RootPathProvider property to create an instance of NancyPathProvider:

```
public class NancyBootstrapper : DefaultNancyBootstrapper
{
    protected override IRootPathProvider RootPathProvider
    {
        get { return new NancyPathProvider(); }
    }
}
```

Topshef lets you easily develop
and deploy a Windows service
from the starting point of a
simple console application.

And when I add Nancy to the OWIN pipeline, I pass an instance of NancyBootstrapper in the options:

```
application.UseNancy(options => options.Bootstrapper = new NancyBootstrapper());
```

Sending Messages

Receiving messages is half the work, but the application still needs a process to send them. This is a process that traditionally would live in an isolated service. In this unified solution I can simply add

an `SmsSender` that launches when the application starts. I'll add this to the `SmsApplication` `Start` method (in a real application, you should add the capability to stop and dispose of this resource):

```
public void Start()
{
    WebApplication = WebApp.Start<WebPipeline>("http://localhost:5000");
    new SmsSender().Start();
}
```

Inside the `Start` method on `SmsSender`, I start a long-running task to send messages:

```
public class SmsSender
{
    public void Start()
    {
        Task.Factory.StartNew(SendMessages, TaskCreationOptions.LongRunning);
    }
}
```

When the `WebApi` send action receives a message, it adds it to a message queue that's a blocking collection. I create the `SendMessages` method to block until messages arrive. This is possible thanks to the abstractions behind `GetConsumingEnumerable`. When a set of messages arrives, it immediately starts sending them:

```
private static void SendMessages()
{
    foreach (var message in MessageQueue.Messages.GetConsumingEnumerable())
    {
        Console.WriteLine("Sending: " + message.Text);
    }
}
```

It would be trivial to spin up multiple instances of `SmsSender` to expand the capacity to send messages. In a real application, you'd want to pass a `CancellationToken` to `GetConsumingEnumerable` to safely stop the enumeration. If you want to learn more about blocking collections, you'll find good information at bit.ly/QgiCM7 and bit.ly/1m6sqll.

Easy, Breezy Deploys

Developing a combined service and Web application is pretty simple and straightforward, thanks to Katana and Topshelf. One of the awesome benefits of this powerful combination is a ridiculously simple deployment process. I'm going to show you a simple two-step deployment using `psake` (github.com/psake/psake). This isn't meant to be a robust script for actual production use; I just want to demonstrate how truly simple the process is, regardless of what tool you use.

The first step is to build the application. I create a build task that will call `msbuild` with the path to the solution and create a release build (the output will end up in `bin/Release`):

```
properties {
    $solution_file = "Sms.sln"
}

task build {
    exec { msbuild $solution_file /t:Clean /t:Build
    /p:Configuration=Release /v:q }
}
```

The second step is to deploy the application as a service. I create a deploy task that depends on the build task and declare a delivery directory to hold a path to the installation location. For simplicity I just deploy to a local directory. Then, I create an executable variable to point to the console application executable in the delivery directory:

```
task deploy -depends build {
    $delivery_directory = "C:\delivery"
    $executable = join-path $delivery_directory 'Sms.exe'
```

First, the deploy task will check if the delivery directory exists. If it finds a delivery directory, it will assume the service is already deployed. In this case, the deploy task will uninstall the service and remove the delivery directory:

```
if (test-path $delivery_directory) {
    exec { & $executable uninstall }
    rd $delivery_directory -rec -force
}
```

Next, the deploy task copies the build output to the delivery directory to deploy the new code, and then copies the views and static folders into the delivery directory:

```
copy-item 'Sms\bin\Release' $delivery_directory -force -recurse -verbose
copy-item 'Sms\views' $delivery_directory -force -recurse -verbose
copy-item 'Sms\static' $delivery_directory -force -recurse -verbose
```

Finally, the deploy task will install and start the service:

```
exec { & $executable install start }
```

When you deploy the service, make sure your `IsDevelopment` implementation returns `false` or you'll get an `Access Denied` exception if the file server can't find the static folder. Also, sometimes reinstalling the service on each deploy can be problematic. Another tactic is to stop, update and then start the service if it's already installed.

Developing a combined service
and Web application is pretty
simple and straightforward,
thanks to Katana and Topshelf.

As you can see, the deployment is ridiculously simple. IIS and `InstallUtil` are completely removed from the equation; there's one deployment process instead of two; and there's no need to worry about how the Web and service layer will communicate. This deploy task can be run repeatedly as you build your unified Web and service application!

Looking Forward

The best way to determine if this combined model will work for you is to find a low-risk project and try it out. It's so simple to develop and deploy an application with this architecture. There's going to be an occasional learning curve (if you use Nancy for MVC, for example). But the great thing about using OWIN, even if the combined approach doesn't work out, is that you can still host the OWIN pipeline inside IIS using the ASP.NET host (Microsoft.Owin.Host.SystemWeb). Give it a try and see what you think. ■

Wes McClure leverages his expertise to help clients rapidly deliver high-quality software to exponentially increase the value they create for customers. He enjoys speaking about everything related to software development, is a *Pluralsight* author and writes about his experiences at devblog.wesmcclure.com. Reach him at wes.mcclure@gmail.com.

THANKS to the following technical experts for reviewing this article:

Howard Dierking (Microsoft), Damian Hickey, Chris Patterson (RelayHealth), Chris Ross (Microsoft) and Travis Smith

Visual Studio[®] LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

WASHINGTON, D.C.

October 6 – 9, 2014

Washington Marriott at Metro Center



**TO BOLDLY CODE
WHERE NO VISUAL STUDIO LIVE! HAS CODED BEFORE**

That's right, we're transporting Visual Studio Live! to our nation's capital for the first time in 21 years. From October 6 – 9, 2014, developers, software architects, engineers and designers will gather for 4 days of cutting-edge education on the Microsoft Platform.

The Washington Marriott Metro Center, our headquarters, is centrally located in the heart of it all: the Mall and National Archives to the South; the White House the West; Mount Vernon Square to the North; and Metro Center to the East. You can explore this country's rich history just steps from the front door – all while sharpening your .NET skills and your ability to build better apps!

SUPPORTED BY

Microsoft



Visual Studio

msdn
magazine

Visual Studio
MAGAZINE

PRODUCED BY

1105 MEDIA[®]



YOUR GUIDE TO THE .NET DEVELOPMENT UNIVERSE



REGISTER TODAY AND SAVE \$300!

Use promo code DCJUN2 by August 13



Scan the QR code to
register or for more
event details.

Tracks Include:

- Visual Studio/.NET Framework
- Windows Client
- JavaScript/HTML5 Client
- ASP.NET
- Cloud Computing
- Windows Phone
- Cross-Platform Mobile Development
- SharePoint
- SQL Server

CONNECT WITH VISUAL STUDIO LIVE!



twitter.com/vslive – @VSLive



facebook.com – Search “VSLive”



linkedin.com – Join the
“Visual Studio Live” group!

vslive.com/dc

The MVVM Light Messenger In-Depth

Laurent Bugnion

This series about the Model-View-ViewModel (MVVM) pattern and the MVVM Light Toolkit has covered quite a lot of ground since I started almost a year ago, from the use of IOC containers in MVVM applications to ways to handle cross-thread access and the DispatcherHelper component of MVVM Light. I also talked about commanding (with the RelayCommand and EventToCommand), view services, such as Navigation and Dialog services, and briefly discussed the Messenger component.

The Messenger component is actually quite a powerful element of the MVVM Light Toolkit, one that often seduces developers thanks to its ease of use, but has also sparked some controversy because of the risks it can create if it's misused. This component deserves its own article explaining how it works, what the risks are and the scenarios for which it makes the most sense.

This article discusses:

- How the MVVM Light Toolkit simplifies messaging
- Cross-thread access
- Avoiding memory leaks and other risks
- Filtering messages
- Using a dedicated message type

Technologies discussed:

MVVM Light Toolkit, Visual Studio, Windows Phone, Windows Presentation Foundation, Windows Runtime

In this article, I'll discuss the general principles behind the Messenger's implementation, and look at why this implementation is easier to use than more traditional approaches. I'll also explore how this approach can impact memory if certain precautions aren't taken. Finally, I'll discuss the MVVM Light Messenger itself in greater detail, in particular some of the built-in messages and their use.

Instead of using words such as "event aggregation," which are hard to define, the talk is about messaging, which is fairly easy to understand.

Event Aggregation and Messenger Simplifications

Systems like the Messenger are sometimes named event buses or event aggregators. Such components connect a sender and a receiver (sometimes called "publisher" and "subscriber," respectively). When MVVM Light was created, many messaging systems required the receiver or the sender to implement specific methods. For instance, there might have been an IReceiver interface that specified a Receive method, and to register with the messaging system,

an object would've had to implement this interface. This kind of constraint was annoying, because it limited who could actually use the messaging system. For example, if you were using a third-party assembly, you couldn't register an instance from this library with the messaging system, because you didn't have access to the code and couldn't modify the third-party class to implement `IReceiver`.

The MVVM Light Messenger was created with the intention of streamlining this scenario with a simple premise: Any object can be a receiver; any object can be a sender; any object can be a message.

One thing the Messenger doesn't do is watch on which thread a message is sent.

The vocabulary, too, was simplified. Instead of using words such as “event aggregation,” which are hard to define, the talk is about messaging, which is fairly easy to understand. The subscriber becomes the receiver and the publisher becomes the sender. Instead of events, there are messages. These simplifications in the language, together with the simplified implementation, make it easier to get started with the Messenger and understand how it works.

For example, consider the code in **Figure 1**. As you can see, the MVVM Light Messenger is being used in two separate objects. The Registration object sends a message to all the RegisteredUser instances. This kind of scenario can be implemented in multiple ways, and the Messenger might not always be the best solution. But, depending on your architecture, it could be a very good tool to implement this feature, especially if the sender and the receiver

Figure 1 Sending and Receiving a Message

```
public class Registration
{
    public void SendUpdate()
    {
        var info = new RegistrationInfo
        {
            // ... Some properties
        };

        Messenger.Default.Send(info);
    }
}

public class RegisteredUser
{
    public RegisteredUser()
    {
        Messenger.Default.Register<RegistrationInfo>(
            this,
            HandleRegistrationInfo);
    }

    private void HandleRegistrationInfo(RegistrationInfo info)
    {
        // Update registered user info
    }
}

public class RegistrationInfo
{
    // ... Some properties
}
```

Figure 2 Registering with Named Methods or Lambdas

```
public UserControl()
{
    InitializeComponent();

    // Registering with named methods ----

    Loaded += Figure2ControlLoaded;

    Messenger.Default.Register<AnyMessage>(
        this,
        HandleAnyMessage);

    // Registering with anonymous lambdas ----

    Loaded += (s, e) =>
    {
        // Do something
    };

    Messenger.Default.Register<AnyMessage>(
        this,
        message =>
        {
            // Do something
        });
}

private void HandleAnyMessage(AnyMessage message)
{
    // Do something
}

private void Figure2ControlLoaded (object sender, RoutedEventArgs e)
{
    // Do something
}
```

are in parts of the application that should remain decoupled. Notice how the Registration instance doesn't send to RegisteredUser instances explicitly. Instead, it broadcasts the message through the Messenger. Any instance can register for this type of message and be notified when it's sent. In this example, the message sent is a RegistrationInfo instance. However, any type of message can be sent, from simple values (int, bool and so on) to dedicated message objects. Later I'll discuss using messages and review some of the built-in message types in MVVM Light.

The code in **Figure 1** shows that registering for a message type (RegistrationInfo) is done through a delegate (HandleRegistrationInfo). This is a common mechanism in the Microsoft .NET Framework. For instance, registering an event handler in C# is also done by passing a delegate to the event, either a named method or an anonymous lambda expression. Similarly, you can use named methods or anonymous lambdas to register a receiver with the Messenger, as shown in **Figure 2**.

Cross-Thread Access

One thing the Messenger doesn't do is watch on which thread a message is sent. If you read my previous article, “Multithreading and Dispatching in MVVM Applications” (bit.ly/1mgZ0Cb), you know that some precautions need to be taken when an object running on a thread attempts to access an object belonging to another thread. This issue often arises between a background thread and a control owned by the UI thread. In the previous article, you saw how the MVVM Light DispatcherHelper can be used to “dispatch” the operation on the UI thread and avoid the cross-thread access exception.

Figure 3 Strong Reference Between Instances

```
public class Setup
{
    private First _first = new First();
    private Second _second = new Second();

    public void InitializeObjects()
    {
        _first.AddRelationTo(_second);
    }

    public void Cleanup()
    {
        _second = null;

        // Even though this is set to null, the Second instance is
        // still kept in memory because the reference count isn't
        // zero (there's still a reference in _first).
    }
}

public class First
{
    private object _another;

    public void AddRelationTo(object another)
    {
        _another = another;
    }
}

public class Second
{
}
```

Some event aggregators let you automatically dispatch messages sent to the UI thread. The MVVM Light Messenger never does that, however, because of the desire to simplify the Messenger API. Adding an option to automatically dispatch the messages to the UI thread would add more parameters to the registration methods. Moreover, it would make the dispatching less explicit and possibly more difficult for less-experienced developers to understand what's happening under the covers.

Instead, you should explicitly dispatch messages to the UI thread if needed. The best way to do that is to use the MVVM Light DispatcherHelper. As shown in the previous article, the `CheckBeginInvokeOnUI` method will dispatch the operation only if necessary. If the Messenger is already running on the UI thread, the message can be distributed immediately without dispatching:

```
public void RunOnBackgroundThread()
{
    // Do some background operation

    DispatcherHelper.CheckBeginInvokeOnUI(
        () =>
        {
            Messenger.Default.Send(new ConfirmationMessage());
        });
}
```

Memory Handling

Every system that allows objects to communicate without knowing each other faces the difficult task of having to save a reference to the recipients of the message. For example, consider that the .NET event handling system can create strong references between an object that raises an event, and an object that subscribes to the event. The code in **Figure 3** creates a strong link between `_first` and `_second`. What this means is that if the `Cleanup` method

is called, and `_second` is set to null, the garbage collector can't remove it from memory, because `_first` still has a reference to it. The garbage collector relies on counting the references to an object to know if it can be removed from memory, and this doesn't happen for the `Second` instance, so a memory leak is created. Over time, this can cause a lot of issues; the application might slow down significantly, and eventually it can even crash.

Every system that allows objects to communicate without knowing each other faces the difficult task of having to save a reference to the recipients of the message.

To mitigate this, .NET developers came up with the `WeakReference` object. This class allows a reference to an object to be stored in a “weak” manner. If all other references to that object are set to null, the garbage collector can still collect the object, even though there's a `WeakReference` using it. This is very convenient, and when

Figure 4 Using WeakReference Instances

```
public class SuperSimpleMessenger
{
    private readonly List<WeakReference> _receivers
        = new List<WeakReference>();

    public void Register(IReceiver receiver)
    {
        _receivers.Add(new WeakReference(receiver));
    }

    public void Send(object message)
    {
        // Locking the receivers to avoid multithreaded issues.
        lock (_receivers)
        {
            var toRemove = new List<WeakReference>();

            foreach (var reference in _receivers.ToList())
            {
                if (reference.IsAlive)
                {
                    ((IReceiver)reference.Target).Receive(message);
                }
                else
                {
                    toRemove.Add(reference);
                }
            }

            // Prune dead references.
            // Do this in another loop to avoid an exception
            // when modifying a collection currently iterated.
            foreach (var dead in toRemove)
            {
                _receivers.Remove(dead);
            }
        }
    }
}
```

facebook



Microsoft
SharePoint 2010



Linked in



twitter

SEE THE WORLD AS A DATABASE

ADO.NET ▪ JDBC ▪ ODBC ▪ SQL SSIS ▪ ODATA
MYSQL ▪ EXCEL ▪ POWERSHELL



Microsoft
SQL Server

Linked in

SAP

OData
Open Data Protocol

Salesforce



facebook



Microsoft
SharePoint 2010

amazon
web services

Microsoft
Visual Studio



ODBC

Microsoft
SQL Server

Microsoft
Excel

Microsoft
BizTalk

MySQL

OData

Work With Relational Data, Not Complex APIs or Services

Whether you are a developer using ADO.NET, JDBC, OData, or MySQL, or a systems integrator working with SQL Server or Biztalk, or even an information worker familiar with ODBC or Excel – our products give you bi-directional access to live data through easy-to-use technologies that you are already familiar with. If you can connect to a database, then you will already know how to connect to Salesforce, SAP, SharePoint, Dynamics CRM, Google Apps, QuickBooks, and much more!



Give RSSBus a try today and see what mean:

visit us online at www.rssbus.com to learn more or download a free trial.

rssbus

INTEGRATION YOUR WAY

Figure 5 Risk of Memory Leak Without Unregistration

Visibility	WPF	Silverlight	Windows Phone 8	Windows Runtime
Static	no risk	no risk	no risk	no risk
Public	no risk	no risk	no risk	no risk
Internal	no risk	risk	risk	no risk
Private	no risk	risk	risk	no risk
Anonymous Lambda	no risk	risk	risk	no risk

used wisely, it can alleviate the memory leaks issue, though it doesn't always solve all the issues. To illustrate this, **Figure 4** shows a simple communication system in which the SimpleMessenger object stores the reference to the Receiver in a WeakReference. Notice the check to the IsAlive property before the message is processed. If the Receiver has been deleted and garbage collected before, the IsAlive property will be false. This is a sign that the WeakReference isn't valid anymore, and should be removed.

The MVVM Light Messenger is built on roughly the same principle, although it is, of course, quite a bit more complex! Notably, because the Messenger doesn't require the Receiver to implement any given interface, it needs to store a reference to the method (the callback) that will be used to transmit the message. In Windows Presentation Foundation (WPF) and the Windows Runtime, this isn't a problem. In Silverlight and Windows Phone, however, the framework is more secure and the APIs prevent certain operations from happening. One of these restrictions hits the Messenger system in certain cases.

To understand this, you need to know what kind of methods can be registered to handle messages. To summarize, a receiving method can be static, which is never an issue; or it can be an instance method, in which case you differentiate between public, internal

and private. In many cases, a receiving method is an anonymous lambda expression, which is the same as a private method.

When a method is static or public, there's no risk of creating a memory leak. When the handling method is internal or private (or an anonymous lambda), it can be a risk in Silverlight and Windows Phone. Unfortunately, in these cases there's no way for the Messenger to use a WeakReference. Again, this is not a problem in WPF or the Windows Runtime. **Figure 5** summarizes this information.

Though the MVVM Light Messenger is a very powerful and versatile component, it's important to keep in mind that there are some risks in using it.

It's important to note that even if there's a risk as indicated in **Figure 5**, failing to unregister doesn't always create a memory leak. That said, to make sure no memory leak is caused, it's good practice to explicitly unregister the receivers from the Messenger when they aren't needed anymore. This can be done using the Unregister method. Note that there are multiple overloads of Unregister. A receiver can be completely unregistered from the Messenger, or you can select to unregister only one given method, but to keep others active.

Other Risks When Using the Messenger

As I noted, though the MVVM Light Messenger is a very powerful and versatile component, it's important to keep in mind that there are

Figure 6 Using the Default Messenger and Checking the Sender

```
public class FirstViewModel
{
    public FirstViewModel()
    {
        Messenger.Default.Register<NotificationMessage>(
            this,
            message =>
            {
                if (message.Sender is MainViewModel)
                {
                    // This message is for me.
                }
            });
    }
}

public class SecondViewModel
{
    public SecondViewModel()
    {
        Messenger.Default.Register<NotificationMessage>(
            this,
            message =>
            {
                if (message.Sender is SettingsViewModel)
                {
                    // This message is for me
                }
            });
    }
}
```

Figure 7 Using a Private Messenger

```
public class MainViewModel
{
    private Messenger _privateMessenger;

    public MainViewModel()
    {
        _privateMessenger = new Messenger();
        SimpleIoc.Default.Register(() => _privateMessenger, "PrivateMessenger");
    }

    public void Update()
    {
        _privateMessenger.Send(new NotificationMessage("DoSomething"));
    }
}

public class FirstViewModel
{
    public FirstViewModel()
    {
        var messenger
            = SimpleIoc.Default.GetInstance<Messenger>("PrivateMessenger");

        messenger.Register<NotificationMessage>(
            this,
            message =>
            {
                // This message is for me.
            });
    }
}
```



Some things are just better together.

Make your Windows Azure environment more appetizing with New Relic.



Get more visibility into your entire Windows Azure environment.
Monitor all your Virtual Machines, Mobile Services, and Web Sites
- all from one powerful dashboard.

newrelic.com/azure

Figure 8 Different Communication Channels with Tokens

```
public class MainViewModel
{
    public static readonly Guid Token = Guid.NewGuid();

    public void Update()
    {
        Messenger.Default.Send(new NotificationMessage("DoSomething"),
            Token);
    }
}

public class FirstViewModel
{
    public FirstViewModel()
    {
        Messenger.Default.Register<NotificationMessage>(
            this,
            MainViewModel.Token,
            message =>
            {
                // This message is for me.
            });
    }
}
```

some risks in using it. I already mentioned potential memory leaks in Silverlight and Windows Phone. Another risk is less technical: Using the Messenger decouples the objects so much that it can be difficult to understand exactly what's happening when a message is sent and received. For a less-experienced developer who has never used an event bus before, it can be difficult to follow the flow of operations. For instance, if you're stepping into a method's call, and this method calls the Messenger.Send method, the flow of the debugging is lost unless you know to search for the corresponding Messenger.Receive method and to place a breakpoint there. That said, the Messenger operations are synchronous, and if you understand how the Messenger works, it's still possible to debug this flow.

I tend to use the Messenger as a "last resort," when more conventional programming techniques are either impossible or cause too many dependencies between parts of the application I want to keep as decoupled as possible. Sometimes, however, it's preferable to use other tools, such as an IOC container and services to achieve similar results in a more explicit manner. I talked about IOC and view services in the first article of this series (bit.ly/1m9HTBX).

One or Multiple Messengers

One of the advantages of messaging systems such as the MVVM Light Messenger is that they can be used even across assemblies—in plug-in scenarios, for example. This is a common architecture for building large applications, especially in WPF. But a plug-in system can also be useful for smaller apps, to easily add new features without having to recompile the main part, for example. As soon as a DLL is loaded in the application's AppDomain, the classes it contains can use the MVVM Light Messenger to communicate with any other component in the same application. This is very powerful, especially when the main application doesn't know how many sub-components are loaded, which is typically the case in a plug-in-based application.

Typically, an application needs only a single Messenger instance to cover all communication. The static instance stored in the Messenger.Default property is probably all you need. However, you

can create new Messenger instances if needed. In such cases, each Messenger acts as a separate communication channel. This can be useful if you want to make sure a given object never receives a message not intended for it. In the code in **Figure 6**, for example, two classes register for the same message type. When the message is received, both instances need to perform some checks to see what the message does.

One of the advantages of messaging systems such as the MVVM Light Messenger is that they can be used even across assemblies—in plug-in scenarios, for example.

Figure 7 shows an implementation with a private Messenger instance. In this case, the SecondViewModel will never receive the message, because it subscribes to a different instance of Messenger and listens to a different channel.

Another way to avoid sending a given message to a particular receiver is to use tokens, as shown in **Figure 8**. This is a kind of contract between a sender and a receiver. Typically, a token is a

Figure 9 Using a Message Type to Define Context

```
public class Sender
{
    public void SendBoolean()
    {
        Messenger.Default.Send(true);
    }

    public void SendNotification()
    {
        Messenger.Default.Send(
            new NotificationMessage<bool>(true, Notifications.PlayPause));
    }
}

public class Receiver
{
    public Receiver()
    {
        Messenger.Default.Register<bool>(
            this,
            b =>
            {
                // Not quite sure what to do with this boolean.
            });

        Messenger.Default.Register<NotificationMessage<bool>>(
            this,
            message =>
            {
                if (message.Notification == Notifications.PlayPause)
                {
                    // Do something with message.Content.
                    Debug.WriteLine(message.Notification + ":" + message.Content);
                }
            });
    }
}
```


unique identifier such as a GUID, but it could be any object. If a sender and a receiver both use the same token, a private communication channel opens between the two objects. In this scenario, the SecondViewModel that didn't use the token will never be notified that a message is being sent. The main advantage is that the receiver doesn't need to write logic to make sure the message was really intended for it. Instead, the Messenger filters out messages based on the token.

Using Messages

Tokens are a nice way to filter messages, but this doesn't change the fact that a message should carry some context in order to be understood. For example, you can use the Send and Receive methods with Boolean content, as shown in **Figure 9**. But if multiple senders send Boolean messages, how is a receiver supposed to know who the message was intended for and what to do with it? This is why it's better to use a dedicated message type in order to make the context clear.

Figure 9 also shows a specific message type being used. `NotificationMessage<T>` is one of the most commonly used message types built into the MVVM Light Toolkit, and it allows any content (in this case, a Boolean) to be sent together with a notification string. Typically, the notification is a unique string defined in a static class called Notifications. This allows sending instructions together with the message.

Figure 10 Sending a `PropertyChangedMessage`

```
public class BankViewModel : ViewModelBase
{
    public const string BalancePropertyName = "Balance";
    private double _balance;

    public double Balance
    {
        get
        {
            return _balance;
        }

        set
        {
            if (Math.Abs(_balance - value) < 0.001)
            {
                return;
            }

            var oldValue = _balance;
            _balance = value;
            RaisePropertyChanged(BalancePropertyName, oldValue, value, true);
        }
    }
}

public class Receiver
{
    public Receiver()
    {
        Messenger.Default.Register<PropertyChangedMessage<double>>(
            this,
            message =>
            {
                if (message.PropertyName == BankViewModel.BalancePropertyName)
                {
                    Debug.WriteLine(
                        message.OldValue + " --> " + message.NewValue);
                }
            }
        );
    }
}
```

Of course, it's also possible to derive from `NotificationMessage<T>`; to use a different built-in message type; or to implement your own message types. The MVVM Light Toolkit contains a `MessageBase` class that can be derived for this purpose, although it's absolutely not compulsory to use this in your code.

Typically, an application needs only a single Messenger instance to cover all communication.

Another built-in message type is the `PropertyChangedMessage<T>`. This is especially useful in relation to the `ObservableObject` and the `ViewModelBase` class that's typically used as the base class for objects involved in binding operations. These classes are implementations of the `INotifyPropertyChanged` interface, which is crucial in MVVM applications with data binding. For example, in the code in **Figure 10**, the `BankAccountViewModel` defines an observable property named `Balance`. When this property changes, the `RaisePropertyChanged` method takes a Boolean parameter that causes the `ViewModelBase` class to "broadcast" a `PropertyChangedMessage` with information about this property, such as its name, the old value and the new value. Another object can subscribe to this message type, and react accordingly.

There are other built-in messages in MVVM Light that are useful in various scenarios. In addition, the infrastructure to build your own custom messages is available. In essence, the idea is to make the life of the receivers easier by providing sufficient context for them to know what to do with the content of the message.

Wrapping Up

The Messenger has proven quite useful in many scenarios that would be hard to implement without a completely decoupled messaging solution. However, it's an advanced tool and should be used with care to avoid creating confusing code that could be hard to debug and maintain later.

This article rounds out the presentation of the MVVM Light Toolkit components. It's an exciting time for .NET developers, with the ability to use the same tools and techniques on multiple XAML-based platforms. With MVVM Light, you can share code among WPF, the Windows Runtime, Windows Phone, Silverlight—and even the Xamarin platforms for Android and iOS. I hope you've found this series of articles useful for understanding how MVVM Light can help you to develop your applications efficiently, while making it easy to design, test and maintain those applications. ■

LAURENT BUGNION is senior director for IdentityMine Inc., a Microsoft partner working with technologies such as Windows Presentation Foundation, Silverlight, PixelSense, Kinect, Windows 8, Windows Phone and UX. He's based in Zurich, Switzerland. He is also a Microsoft MVP and a Microsoft Regional Director.

THANKS to the following Microsoft technical expert for reviewing this article:
Jeffrey Ferman

Using JSLink with SharePoint 2013

Pritam Baldota

Working with the SharePoint UI has always been something of a challenge for developers. However, a new feature in SharePoint 2013 called JSLink offloads the burden of using XSLT and provides a much easier and more responsive way to display custom fields on the client. JSLink is a property that controls rendering of fields, items and even Web Parts through a JavaScript file.

This article will explore the use of JSLink with the help of two demo scenarios. The first scenario will demonstrate how you can use color-coded messages to indicate task completion, for example, as shown in **Figure 1**.

The second scenario is more complex. It will demonstrate how to implement an image gallery that provides a callout for each image to show metadata and allow downloading at different resolutions, as shown in **Figure 2**.

This article discusses:

- How the JSLink property improves client-side rendering
- Various ways to set the JSLink JavaScript file reference
- Two scenarios that demonstrate the use of JSLink
- Customizing the New and Edit form fields

Technologies discussed:

SharePoint 2013, JavaScript, Windows PowerShell

Code download available at:

msdn.microsoft.com/magazine/msdnmag0614

I'll also delve into customizing the New and Edit form fields. Before I get started, though, I'll take a look at the basics of client-side rendering in SharePoint 2013 and how JSLink adds value to it.

JSLink is a property that can be used with fields, Web Parts, list forms and content types.

Client-Side Rendering

Client-side rendering refers to the displaying of data on a page using technologies that operate on the client, such as JavaScript, HTML and CSS. Because the technology runs in the client's browser, it's more responsive and efficient, thereby reducing the load on the Web server. Earlier versions of SharePoint (2010, 2007) used XSLT to style elements, which is generally more complex to work with and slower in performance compared with JavaScript. SharePoint 2013 still supports XSLT, but has two additional techniques for customizing results on the client side—display templates, which define the way a SharePoint 2013 Search Web Part renders results (see bit.ly/1i5fM6k for more information) and, the subject of this article's focus, JSLink.

JSLink is a property that can be used with fields, Web Parts, list forms and content types. With this property, you can add JavaScript

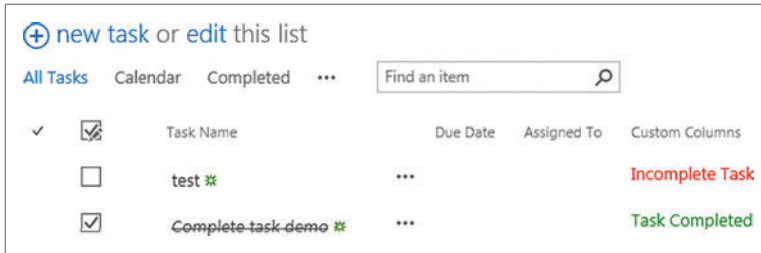


Figure 1 Using a Custom Column for a Task List to Show Status

files, opening up broad possibilities for customization. Each JavaScript file you add is prefaced with a SharePoint token and separated by the pipe symbol (|), like this: ~site/style library/mycustom.js|~site/style library/mycustom2.js. If the JavaScript files don't contain any relevant rendering code, default rendering of the element is applied.

SharePoint provides several tokens (static and dynamic) that are useful for constructing a context-specific URL. Here are some important SharePoint tokens for dynamic URL construction:

- ~site—refers to the URL of the current Web site.
- ~sitecollection—refers to the URL of the parent site collection of the current Web site.
- ~layouts—refers to _layouts/15 with respect to the Web application.
- ~sitecollectionlayouts—refers to the layouts folder in the current site collection (such as /sites/mysite/_layouts/15).
- ~sitelayouts—refers to the layouts folder in the current site (such as site/mysite/mysubsite/_layouts/15).

SharePoint has more tokens for URL construction. To learn more about URL strings and tokens, see the Dev Center at bit.ly/1lpYuAP.

JSLink vs. XSL/XSLT

Client-side rendering using JSLink has a number of advantages over XSL/XSLT. In the first place, it uses JavaScript, with which most Web developers are already comfortable. XSLT is somewhat more complex to develop and debug, so JSLink can reduce development time with no loss of accuracy.

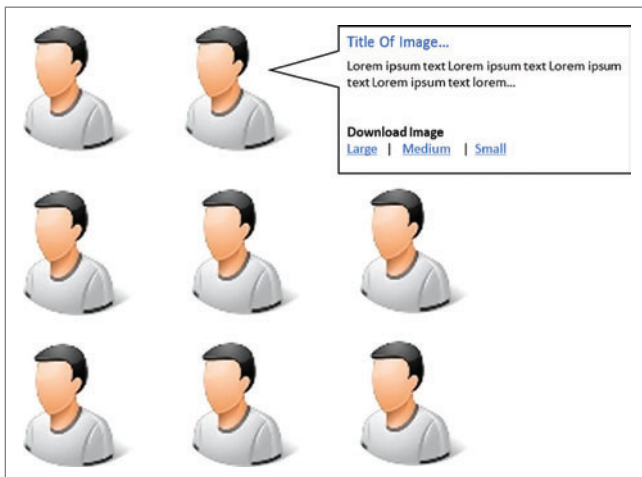


Figure 2 Multiple Resolutions of an Image Available for Download

Rendering a view on the client using JavaScript, HTML, and CSS avoids unnecessary loads on the server, improving overall performance and reducing page response times. Client-side processing makes the UI highly responsive.

Furthermore, you can customize all or part of a view using JSLink. For example, if you want to customize just a specific field, you can tailor the rendering logic for that field only; the rest of the view will be rendered using the default logic. With JSLink, you can use any

valid JavaScript, including external plug-ins such as jQuery, in combination with HTML and CSS.

Of course, every technology has some disadvantages, and so does JSLink. For example, if a user has JavaScript blocked in his browser, JSLink won't work. Server-side rendering with XSLT will still show the same experience, but performance might suffer.

XSLT is somewhat more complex to develop and debug, so JSLink can reduce development time with no loss of accuracy.

Performance might also suffer if a user's browser or system is old or underpowered because it may take more time to execute the script.

Finally, crawlers are unlikely to understand the dynamic content generated by AJAX/JavaScript; they require static data rendered with HTML. So, if you have a public-facing site, JSLink may not be your best choice.

Setting the JSLink Reference

The JSLink JavaScript file reference can be set in multiple ways using the server object model, Windows PowerShell, Element.xml via Features, the Web Part Properties window and the client object model. Following is some sample code for each approach.

The Server Object Model: To set the JSLink property of, say, List forms, you access the SPForm object using the list's Forms collection and then set the JSLink property to the SPForm object:

```
SPWeb web = SPContext.Current.Web;
SPList list = web.Lists.TryGetList("MyTasks");
if (null != list)
{
    SPForm newForm = list.Forms[PAGETYPE.PAGE_NEWFORM];
    if (null != newForm)
    {
        newForm.JSLink = "~/mycustom.js";
    }
}
```

Windows PowerShell: To set the JSLink property of, for example, a custom field of a list, you access the field object using the list's Fields collection and then set the JSLink property to the field object:

```
$web = Get-SPWeb
$field = $web.Fields["MyCustomField"]
$field.JSLink = "~/layouts/mycustom.js"
$field.Update()
$web.Dispose()
```


The Element.xml file: To set the JSLink property of a custom field of a list, you add a Field node into the Element.xml file:

```
<Field ID="{eb3eed37-961b-41bd-b11c-865c16e47071}"
Name="MyCustomField" DisplayName="Custom Columns"
Type="Text" Required="FALSE" Group="JSLink Demo"
JSLink="~site/style library/JSLinkDemo/jquery-1.10.2.min.js|
~site/style library/JSLinkDemo/customview.js">
</Field>
```

Notice that by using the pipe symbol you can add multiple JavaScript files.

The Web Parts properties dialog: To set the JSLink property of a Web Part, you modify its properties. Go to Web Part | Edit Properties | Miscellaneous and set the JSLink property.

The Client-Side Object Model (CSOM): You can set the JSLink of a field using the CSOM. Note that you can't update the site column property via JavaScript directly, you need to update it using list-associated fields. If you try to update at site column level, you'll get this error:

This functionality is unavailable for fields not associated with a list ...

This code shows how to correctly update the JSLink property for a field in a list via the JavaScript CSOM:

```
fieldCollection = taskList.get_fields();
this.oneField = fieldCollection.
getByInternalNameOrTitle("MyCustomField");
this.oneField.set_description("MyNewFieldDescription");
this.oneField.update();
```

For more information on this example, see the MSDN documentation at bit.ly/1i9rlZR.

Figure 3 Referencing the JavaScript Files

```
<?xml version="1.0" encoding="utf-8"?>
<Elements xmlns="http://schemas.microsoft.com/sharepoint/">
  <Field
    ID="{eb3eed37-961b-41bd-b11c-865c16e47071}"
    Name="MyCustomField"
    DisplayName="Custom Columns"
    Type="Text"
    Required="FALSE"
    Group="JSLink Demo"
    JSLink="~site/style library/JSLinkDemo/jquery-1.10.2.min.js|
~site/style library/JSLinkDemo/customview.js">
  </Field>
</Elements>
```

Figure 4 Displaying Images in Tabular Format

```
function customItem(ctx) {
  // Grid contains 4
  if (ctx.CurrentItemIdx % 4 == 0) {
    // Start new row
    tableData += "<div style='clear:both'></div>"
  }

  tableData += "<div style='margin:5px;border:1px solid #666;float:left;"+
"width:100px;height:100px' onmouseover='\""+
"ShowCallout(this,\" + ctx.CurrentItem.Title + \",\" +
ctx.CurrentItem.FileDirRef + \",\" + ctx.CurrentItem.FileLeafRef
+ \");\"><img src='\" + ctx.CurrentItem.FileRef +
\"' width='100' height='100' /></div>";

  return "";
}

// Custom Footer Method
function customFooter(ctx) {
  // Append Dynamic-generated data to container div
  $("#customImageGalleryContainer").html(tableData);
  return "";
}
```

Working with JSLink

In order to work with JSLink, in the custom JavaScript file you need to override this method:

```
SPClientTemplates.TemplateManager.RegisterTemplateOverrides()
```

This method requires a template object to be passed to it. To define the template object, you must specify a property and a render method for each view (such as View, Add and Edit).

You can set the JSLink of a field using the CSOM.

The template object for registering the method has properties such as Header, Body, Footer, OnPreRender, OnPostRender, Group, Item and Fields, which can be leveraged to override the default rendering logic of a view. For example, to modify the custom fields View, Display, Edit and New, you provide the following information for the template object's Fields property:

```
siteCtx.Templates.Fields = {
  // MyCustomField is the Name of our field
  'MyCustomField': {
    'View': customView,
    'DisplayForm': customDisplayForm,
    'EditForm': customNew,
    'NewForm': customEdit
  }
};
```

The methods referenced in this code, such as customView and customDisplayForm, contain the actual rendering logic for this field.

Finally, you call the RegisterTemplateOverrides method of TemplateManager to apply the custom view, like so:

```
// Register the template to render custom field
SPClientTemplates.TemplateManager.RegisterTemplateOverrides(siteCtx);
```

This method will take care of the rest of the rendering of the view based on your custom logic.

Keep in mind that when you use multiple instances of a list view Web Part on the same page and apply JSLink to one of the instances, the layout of all other list view Web Parts changes also, because internally SharePoint uses common rendering logic. To avoid this problem, you need to make sure the BaseViewID value won't conflict with existing ViewIDs, so change the BaseViewID property of the context before overriding, for example, ctx.BaseViewID = 1000.

Now I'll walk you through the two demonstration scenarios.

Scenario 1: Display Color-Coded Task Completion Status

This scenario shows task completion status using color coding for a Task List—red for an incomplete task and green for a completed task.

In the example, I'll customize the View template for the custom field, as shown here:

```
// View Page custom rendering
function customView(ctx) {
  if (ctx != null && ctx.CurrentItem != null) {
    var percentComplete = parseInt(ctx.CurrentItem.PercentComplete);
    if (percentComplete != 100) {
      return "<span style='color:red'>Incomplete Task</span>";
    }
    else {
      return "<span style='color:green'>Task Completed</span>";
    }
  }
}
```

Visual Studio **LIVE!**
EXPERT SOLUTIONS FOR .NET DEVELOPERS

REDMOND 2014

August 18 – 22 | Microsoft Headquarters | Redmond, WA



**SET YOUR COURSE
FOR 127.0.0.1!**



Topics include:

- Visual Studio/.NET Framework
- Windows Client
- JavaScript/HTML5 Client
- ASP.NET
- Cloud Computing
- Windows Phone
- Cross-Platform Mobile Development
- SharePoint
- SQL Server

**YOUR GUIDE TO THE
.NET DEVELOPMENT
UNIVERSE**

**SAVE
\$400**



Use promo code
REDJUNTI by June 11

vslive.com/redmond

Visual Studio
EXPERT SOLUTIONS FOR .NET DEVELOPERS



REDMOND 2014

August 18 – 22 | Microsoft Headquarters | Redmond, WA



From August 18 – 22, 2014, developers, engineers, software architects and designers will land in Redmond, WA at the idyllic Microsoft headquarters for 5 days of cutting-edge education on the Microsoft Platform.

Come experience code at the source – rub elbows with Microsoft stars, get the inside scoop on what's next, and learn what you need to know now. Over 5 days and 60+ sessions and workshops, you'll explore the .NET Development Universe, receiving the kind of practical, unbiased training you can only get at Visual Studio Live!

vslive.com/redmond

SAVE \$400

vslive.com/redmond

CONNECT WITH VISUAL STUDIO LIVE!



twitter.com/vslive – @VSLive



facebook.com – Search "VSLive"



linkedin.com – Join the "Visual Studio Live" group!



Use promo code
REDJUNTI by June 11

EVENT SPONSOR

Microsoft

PLATINUM SPONSOR



SPONSOR



SUPPORTED BY



PRODUCED BY



Figure 5 The Updated ShowCallout Function

[illegible]

The `customView` method receives the current render context as an input parameter from SharePoint internally. This input context has lots of other properties related to List, View, Current Item and so forth.

The context's `currentItem` property gives access to the current row item of a list. Through this property you can access all fields available in the List. Note that if the Field isn't available in a selected view, you'll get an error when accessing the field value.

To begin, open Visual Studio 2013 and choose a SharePoint 2013 Empty Project.

Step 1: From the Add New Item menu, add a Site Column to the empty project. To create a Site Column, add the Field information in an Element.xml file, as shown in **Figure 3**. As you can see, the JSLink property references two JavaScript files: the JQuery library and a custom view JavaScript file from the Style Library. Note that the hierarchy of files should be maintained from left to right (dependent files should be referenced first).

Step 2: Add another New Item, a Scripts module to store custom JavaScript and other resources files.

Step 3: Add the JQuery Library to the Scripts module. Create a new JavaScript file called CustomView.js.

Step 4: Rename the default Feature from Feature 1 to JSLink-Demo, or create a new Feature that includes the Site Columns and Scripts module. Set the scope to Site as it's the Site Column that will deploy.

Step 5: Deploy the solution. After you deploy the solution, you'll see the Column added to Site Columns by going to Site Settings | Site Columns.

Step 6: Create a task list called MyTasks by adding an app from the Site contents page.

Step 7: Add a custom column to the MyTasks list from the List Settings Menu on the ribbon.

Step 8: Add a custom column as created using Steps 1 to 5 by clicking Add from existing site columns on the List Settings page. Filter the group using JSLink Demo, choose Custom Column and add it to the list.

The context's `currentItem` property gives access to the current row item of a list.

This completes the implementation of the first scenario, which displays a task's completion status. As you can see in **Figure 1**, the custom status shows as either Task Completed in green or Incomplete Task in red.

Scenario 2: Create a Custom Image Gallery with Callouts

In this scenario I'll customize the built-in image gallery rendering to display in a custom tabular format, with callouts to download multiple resolution images.

To do this, I customize the Item, Header and Footer properties of Template Field. The following code sets up the initial step:

```
(function () {  
    var overrideContext = {};  
    overrideContext.ListTemplateType = 109;  
    overrideContext.Templates = {};  
    overrideContext.Templates.Item = customItem;  
    SPClientTemplates.TemplateManager.  
    RegisterTemplateOverrides(overrideContext);  
})();  
  
function customItem() {  
    return "ItemRow";  
}
```

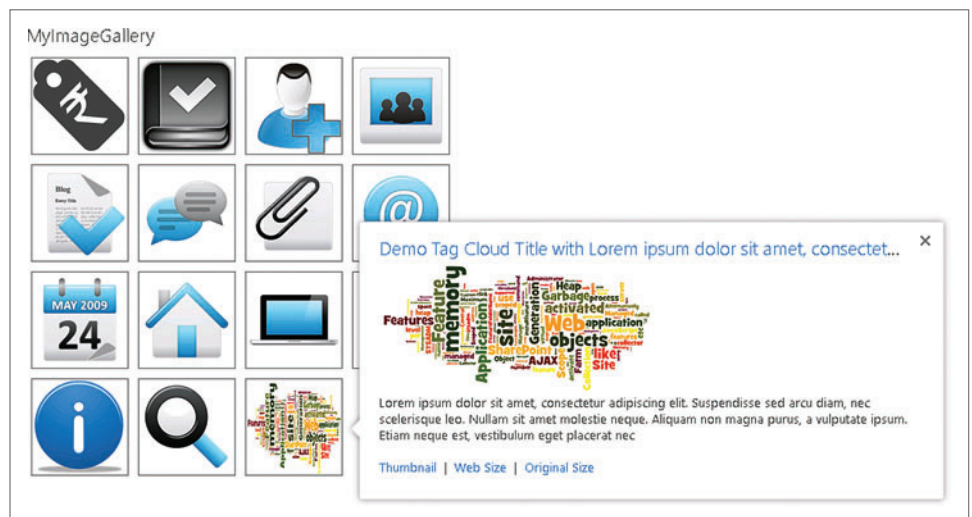


Figure 6 Custom Image Gallery with Callout Showing Metadata

Figure 7 Multi-Column Custom Fields of a List Using JSLink

I've set ListTemplateType to 109, which SharePoint predefines as PictureLibrary. (You'll find all the predefined list template types in SharePoint 2013 at bit.ly/1qE8UiY.) This code will render rows with text returned from the customItem method.

Now, I'll replace the static text with the actual properties of Image (Title, URL and other fields) to start rendering the image. To replace the default new picture link and header columns, I'll customize the header and footer properties to set some HTML text:

```
overrideContext.Templates.Header = "<b>Custom Image Gallery View</b>";
overrideContext.Templates.Footer = "<b>Custom Footer</b>";
function customItem() {
    return "<br /><img src='" + ctx.CurrentItem.FileRef +
        "' width='190' height='190' />";
}
```

I want to display the images in tabular format. To do so, I have to add some CSS and HTML for the layout in the customItem method. First, I add an empty div container to the header template, which will be used later on to hold the actual data from Item and Footer rendering. Within the Item rendering, I create dynamic HTML and store it in a global variable. In the Footer rendering template, I assign the dynamic data to the div container. **Figure 4** shows the complete code to render a tabular format image gallery.

Now I want to display a callout on mouse over for each image, and show additional metadata such as Title, Brief Info and so on, along with different image resolutions for download. SharePoint 2013 provides a callout framework to show contextual information via a callout.js JavaScript file, using the CalloutManager global object. I'll leverage that framework to show default callouts. First, I create a custom method called ShowCallout to determine if there are any existing open callouts; if there are, I will close them using the default closeAll method from the CalloutManager:

```
function ShowCallout(sender, itemId, title, brief, directory, filename) {
    CalloutManager.closeAll();
}
```

Figure 8 Validation Code

```
var validatorContainer = new SPClientForms.ClientValidation.ValidatorSet();
if (formCtx.fieldSchema.Required) {
    validatorContainer.RegisterValidator(
        new SPClientForms.ClientValidation.RequiredValidator());
}

if (validatorContainer._registeredValidators.length > 0) {
    formCtx.registerClientValidator(
        formCtx.fieldName, validatorContainer);
}

formCtx.registerValidationErrorMessageCallback(
    formCtx.fieldName, function (errorResult) {
        SPFormControl_AppendValidationErrorMessage(
            errorContainer, errorResult);
    });
```

Before moving ahead with the implementation, you need to understand how to get images of different resolutions. In SharePoint 2013, by default an image is created in two different sizes when it's uploaded to an image library. For example, if you upload sample.png to library path /imageLibraryName/, SharePoint automatically creates both a thumbnail and a Web-size image: /imageLibraryName/_t/Sample_png.jpg and /imageLibraryName/_w/Sample_png.jpg. In these URLs, _t represents Thumbnail and _w represents Web, and the file extension gets appended to the filename with a prefix underscore (_) separator.

The ShowCallout function uses these URLs for the original image, as well as for images of different resolutions, and stores them in different variables as shown here (Filename and directory are ShowCallout parameters):

```
var fname = filename.replace(".", "_");
var thumbnail = directory + "_t/" + fname + ".jpg";
var medium = directory + "_w/" + fname + ".jpg";
var full = directory + "/" + filename;
```

SharePoint 2013 provides a callout framework to show contextual information via a callout.js JavaScript file, using the CalloutManager global object.

For this scenario I use the CalloutManager.createNewIfNecessary method, which creates the callout only if there's no callout at the target launch point. If there is a callout, the method returns the existing object reference. (To read more about CalloutManager, visit bit.ly/1kXH7uU.) **Figure 5** shows the complete code, and **Figure 6** shows the resulting output.

To apply the custom rendering, create a site page and add the image library List View Web Part. Go to the Edit Web Part properties dialog and set JSLink property to ~/site/Style Library/JSLinkDemo/jquery-1.10.2.min.js|~/site/Style Library/JSLinkDemo/Custom-ImageGallery.js. Click Apply and Save Page. After refreshing the page, you'll see a Gallery like the one shown in **Figure 6**.

Figure 9 Rendering Input Controls Dynamically

```
// Render input fields
var inputFields = "<table>";
...
inputFields += "</table>";

// Get List data from REST API
$.ajax({
    ...
    success: function (data) {
        // Add options to the dynamic dropdown list
    },
    error: function (data) {
        // Error handler
    }
});

return inputFields;
```

Customize a New/Edit List Form Using JSLink

You can customize the New and Edit form fields using a similar approach, though with some variations. Here you need to consider such things as input validation, storing data from input field to list item, displaying stored data from a list item in an input field and so forth.

Next, I'll take a look at the implementation details of a complex, multi-column custom field scenario as shown in **Figure 7**.

Step 1: Using the Element.xml file, create a Note (a custom field for text under Site Columns with the NumLines property set to, say, 1,000 (or whatever suits your needs). This custom field overrides the rendering on the New and Edit form.

```
siteCtx.Templates.Fields = {
  "MyComplexField": {
    'EditForm': customComplexNewOrEdit,
    'NewForm': customComplexNewOrEdit
  }
};
```

I use the same customComplexNewOrEdit method for both view New and Edit.

You can debug JSLink
using browser-based developer
tools such as the Internet
Explorer developer tools,
Firebug and the like.

Step 2: Set up the form context, which is required for setting up validators to read a value from the custom view and to save it back to a list. To set up the form context you use the SPClientTemplates.Utility.GetFormContextForCurrentField method. This method accepts Render Context as parameter, which is provided by SharePoint internally. Here's some sample code showing the new or edit custom render method:

```
function customComplexNewOrEdit(ctx) {
  if (ctx == null || ctx.CurrentFieldValue == null)
    return '';

  var formCtx = SPClientTemplates.Utility.
    GetFormContextForCurrentField(ctx);
  if (formCtx == null || formCtx.fieldSchema == null)
    return '';
}
```

Step 3: After receiving the form context, you need to register callback handlers with the current form to enable validation, get a field's value and save the value into the appropriate field on the custom view generated from JSLink. Every time a user clicks the save button on the List form, SharePoint internally invokes a callback handler attached to the fields using formCtx.registerGetValueCallback(fieldName, callback). This callback handler reads the value from the field before saving it to list item:

```
formCtx.registerGetValueCallback(formCtx.fieldName, function () {
  // Read value from this callback and assign to the field before save
  return "";
});
```

This is an empty callback handler. You have to customize the code in order to read the values from the selected controls on the form and return the string representation of the values.

Step 4: To add validators, you need to first create a validator container in which to register validators for the various fields. I do this using the SPClientForms.ClientValidation.ValidatorSet object. I can use the RegisterValidator method of Validator Container to register multiple validators.

You need to register an error callback handler, as well, to show errors. To do so, I'll use the registerValidationErrorCallback method of form context. This method requires two parameters, the ID of the HTML container element (div in this example) and errorResult, which is returned from SharePoint internally based on validation failure. Error messages will append to the container element provided to this method. In my example I need to add a required field validator, and the complete code for validation is shown in **Figure 8**.

Step 5: You need to add custom HTML controls to represent the field on the new or edit page. I'll have one text box, one dropdown list with dynamic values from another list and one dropdown list with static values. I'll create a table, add these controls to the table and return the HTML as output from the custom Add/Edit method.

To load the data dynamically, I use the REST API to read the data from another list and use it to fill the title in this list. To find out more about the REST API in SharePoint, see bit.ly/1cVNaqA.

Figure 9 shows the code for rendering input controls dynamically.

Step 6: To save a multi-field value to a list, I put all of the fields' values into a single string using the string separator, and create a custom format like (fieldname:value)(fieldname:value), a kind of key-value pair. I'll do the string construction in registerGetValueCallback.

Step 7: To display an existing value in the edit form, I parse the string stored as a key-value pair, saving and assigning the values to respective input controls. This is accomplished using the same custom render method as when constructing custom HTML.

Wrapping Up

You can debug JSLink using browser-based developer tools such as the Internet Explorer developer tools, Firebug and the like. If you put a breakpoint in JavaScript after it renders on the client, you can debug the JavaScript just like C#.NET code. When you access the context in the View method, you can get a reference to the current Item through which you can see the values for each field of the current item. You can use the Internet Explorer developer tools console window to access any property of the current object.

JSLink provides a simple way to customize any SharePoint view using a pure client-side script, without writing a single line of server-side code. You can use any JavaScript-based plug-in with JSLink. Download the sample demo project at msdn.microsoft.com/magazine/msdnmag0614. ■

PRITAM BALDOTA is a SharePoint consultant at Microsoft Services Global Delivery with more than nine years of industry experience. In his leisure time he blogs at pritamaldota.com. Reach him at pritam@pritamaldota.com.

THANKS to the following Microsoft technical experts for reviewing this article:
Sanjay Arora, Subhajt Chatterjee and Paresh Moradiya



Working with the MNIST Image Recognition Data Set

One of the most fascinating topics in the field of machine learning is image recognition (IR). Examples of systems that employ IR include computer login programs that use fingerprint or retinal identification, and airport security systems that scan passenger faces looking for individuals on some sort of wanted list. The MNIST data set is a collection of simple images that can be used to experiment with IR algorithms. This article presents and explains a relatively simple C# program that introduces you to the MNIST data set, which in turn acquaints you with IR concepts.

It's unlikely you'll need to use IR in most software applications, but I think you might find the information in this article useful for four different reasons. First, there's no better way to understand the MNIST data set and IR concepts than by experimenting with actual code. Second, having a basic grasp of IR will help you understand the capabilities and limitations of real, sophisticated IR systems. Third, several of the programming techniques explained in this article can be used for different, more common tasks. And fourth, you might just find IR interesting in its own right.

The best way to see where this article is headed is to take a look at the demo program in **Figure 1**. The demo program is a classic Windows Forms application. The button control labeled Load Images reads into memory a standard image recognition data set called the MNIST data set. The data set consists of 60,000 handwritten digits from 0 through 9 that have been digitized. The demo has the ability to display the currently selected image as a bit-mapped image (on the left of **Figure 1**), and as a matrix of pixel values in hexadecimal form (on the right).

In the sections that follow, I'll walk you through the code for the demo program. Because the demo is a Windows Forms application, much of the code is related to UI functionality, and is contained in multiple files. I focus here on the logic. I refactored the demo code into a single C# source file that's available at msdn.microsoft.com/magazine/msdnmag0614. To compile the download, you can save it on your local computer as `MnistViewer.cs`, then create a new Visual Studio project and add the file to your project. Alternatively, you can launch a Visual Studio command shell (which knows the location of the C# compiler), then navigate to the directory where you saved the download, and issue the command `>csc.exe /target:winexe MnistViewer.cs` to create the executable `MnistViewer.exe`. Before you can run the demo program, you'll need to download and save

two MNIST data files, as I explain in the next section, and edit the demo source code to point to the location of those two files.

This article assumes you have at least intermediate-level skills with C# (or similar language) but doesn't assume you know anything about IR. The demo code makes extensive use of the Microsoft .NET Framework so refactoring the demo code to a non-.NET language such as JavaScript would be a difficult task.

One of the most fascinating topics in the field of machine learning is image recognition.

The terminology used in IR literature tends to vary quite a bit. Image recognition might also be called image classification, pattern recognition, pattern matching or pattern classification. Although these terms do have different meanings, they're sometimes used interchangeably, which can make searching the Internet for relevant information a bit difficult.

The MNIST Data Set

The mixed National Institute of Standards and Technology (MNIST for short) data set was created by IR researchers to act as a benchmark for comparing different IR algorithms. The basic idea is that if you have an IR algorithm or software system you want to test, you can run your algorithm or system against the MNIST data set and compare your results with previously published results for other systems.

The data set consists of a total of 70,000 images; 60,000 training images (used to create an IR model) and 10,000 test images (used to evaluate the accuracy of the model). Each MNIST image is a digitized picture of a single handwritten digit character. Each image is 28 x 28 pixels in size. Each pixel value is between 0, which represents white, and 255, which represents black. Intermediate pixel values represent shades of gray. **Figure 2** shows the first eight images in the training set. The actual digit that corresponds to each image is obvious to humans, but identifying the digits is a very difficult challenge for computers.

Curiously, the training data and the test data are each stored in two files rather than in a single file. One file contains the pixel values for the images and the other contains the label information (0 through

Code download available at msdn.microsoft.com/magazine/msdnmag0614.

9) for the images. Each of the four files also contains header information, and all four files are stored in a binary format that has been compressed using the gzip format.

Notice in **Figure 1**, the demo program uses only the 60,000-item training set. The format of the test set is identical to that of the training set. The primary repository for the MNIST files is currently located at yann.lecun.com/exdb/mnist. The training pixel data is stored in file `train-images-idx3-ubyte.gz` and the training label data is stored in file `train-labels-idx1-ubyte.gz`. To run the demo program, you need to go to the MNIST repository site and download and unzip the two training data files. To unzip the files, I used the free, open source 7-Zip utility.

Creating the MNIST Viewer

To create the MNIST demo program, I launched Visual Studio and created a new C# Windows Forms project named `MnistViewer`. The demo has no significant .NET version dependencies so any version of Visual Studio should work.

After the template code loaded into the Visual Studio editor, I set up the UI controls. I added two `TextBox` controls (`textBox1`, `textBox2`) to hold the paths to the two unzipped training files. I added a `Button` control (`button1`) and gave it a label of `Load Images`. I added two more `TextBox` controls (`textBox3`, `textBox4`) to hold the values of the current image index and the next image index. Using the Visual Studio designer, I set the initial values of these controls to "NA" and "0", respectively.

I added a `ComboBox` control (`comboBox1`) for the image magnification value. Using the designer, I went to the control's `Items` collection and added the strings "1" through "10". I added a second `Button` control (`button2`) and gave it a label of `Display Next`. I added a `PictureBox` control (`pictureBox1`) and set its `BackColor` property to `ControlDark` so that the control's outline could be seen. I set the `PictureBox` size to 280 x 280 to allow a magnification of up to 10 times (recall an MNIST image is 28 x 28 pixels). I added a fifth `TextBox` (`textBox5`) to display the hex values of an image, then set its `Multiline` property to `True` and its `Font` property to `Courier New`, 8.25 pt., and expanded its size to 606 x 412. And, finally, I added a `ListBox` control (`listBox1`) for logging messages.

After placing the UI controls onto the Windows Form, I added three class-scope fields:

```
public partial class Form1 : Form
{
    private string pixelFile =
        @"C:\MnistViewer\train-images.idx3-ubyte";
    private string labelFile =
        @"C:\MnistViewer\train-labels.idx1-ubyte";
    private DigitImage[] trainImages = null;
    ...
}
```

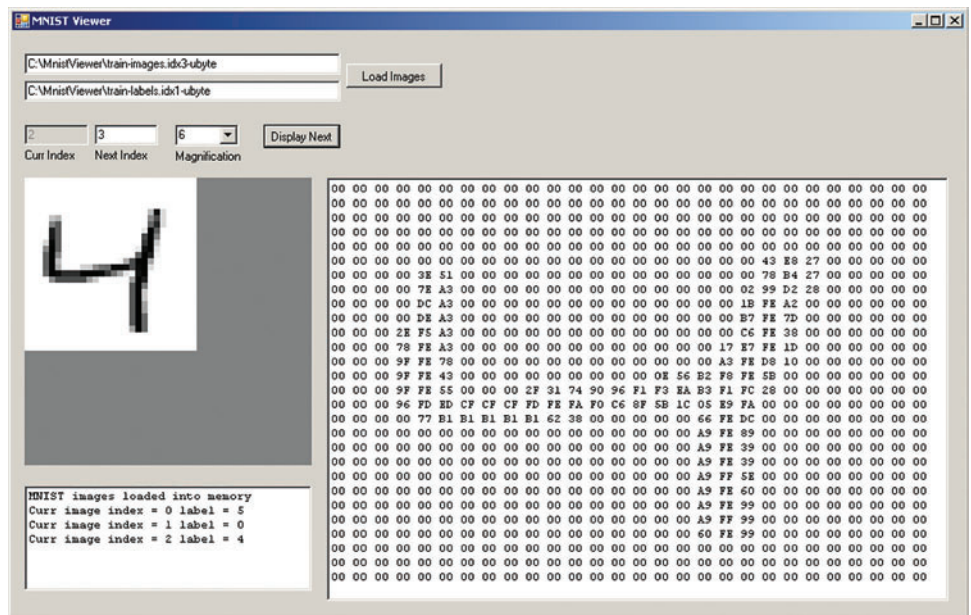


Figure 1 Displaying MNIST Images

The first two strings point to the locations of the unzipped training data files. You'll need to edit these two strings to run the demo. The third field is an array of program-defined `DigitImage` objects.

I edited the `Form` constructor slightly to place the file paths into `textBox1` and `textBox2`, and give the magnification an initial value of 6:

```
public Form1()
{
    InitializeComponent();
    textBox1.Text = pixelFile;
    textBox2.Text = labelFile;
    comboBox1.SelectedItem = "6";
    this.ActiveControl = button1;
}
```

I used the `ActiveControl` property to set the initial focus onto the `button1` control, just for convenience.

Creating a Class to Hold an MNIST Image

I created a small container class to represent a single MNIST image, as shown in **Figure 3**. I named the class `DigitImage` but you might want to rename it to something more specific, such as `MnistImage`.

I declared all class members with public scope for simplicity and removed normal error checking to keep the size of the code small. Fields `width` and `height` could've been omitted because all MNIST images are 28 x 28 pixels, but adding the `width` and `height` fields gives the class more flexibility. Field `pixels` is an array-of-arrays-style matrix. Unlike many languages, C# has a true multidimensional array and you might want to use it instead. Each cell value is type `byte`, which is just an integer value between 0 and 255. Field `label` is also declared as type `byte`, but could've been type `int` or `char` or `string`.

The `DigitImage` class constructor accepts values for `width`, `height`, the `pixels` matrix, and the `label`, and just copies those parameter values to the associated fields. I could've copied the pixel values by reference instead of by value, but that could lead to unwanted side effects if the source pixel values changed.

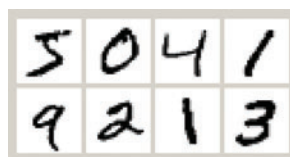


Figure 2 First Eight MNIST Training Images

Loading the MNIST Data

I double-clicked on the button1 control to register its event handler. The event handler farms most of the work to the method LoadData:

```
private void button1_Click(object sender, EventArgs e)
{
    this.pixelFile = textBox1.Text;
    this.labelFile = textBox2.Text;
    this.trainImages = LoadData(pixelFile, labelFile);
    listBox1.Items.Add("MNIST images loaded into memory");
}
```

The LoadData method is listed in **Figure 4**. LoadData opens both the pixel and label files and reads them simultaneously. The method begins by creating a local 28 x 28 matrix of pixel values. The handy .NET BinaryReader class is designed specifically for reading binary files.

The format of the MNIST training pixels file has an initial magic integer (32 bits) that has value 2051, followed by the number of images as an integer, followed by the number of rows and the number of columns as integers, followed by the 60,000 images x 28 x 28 pixels = 47,040,000 byte values. So, after opening the binary files, the first four integers are read using the ReadInt32 method. For example, the number of images is read by:

```
int imageCount = brImages.ReadInt32();
imageCount = ReverseBytes(imageCount);
```

Interestingly, the MNIST files store integer values in big endian format (used by some non-Intel processors) rather than the more usual little endian format that's most commonly used on hardware that runs Microsoft software. So, if you're using normal PC-style hardware, to view or use any of the integer values, they must be converted from big endian to little endian. This means reversing the order of the four bytes that make up the integer. For example, the magic number 2051 in big endian form is:

```
00000011 00001000 00000000 00000000
```

That same value stored in little endian form is:

```
00000000 00000000 00001000 00000011
```

Notice it's the four bytes that must be reversed, rather than the entire 32-bit sequence. There are many ways to reverse bytes. I used a high-level approach that leverages the .NET BitConverter class, rather than using a low-level, bit-manipulation approach:

```
public static int ReverseBytes(int v)
{
    byte[] intAsBytes = BitConverter.GetBytes(v);
    Array.Reverse(intAsBytes);
    return BitConverter.ToInt32(intAsBytes, 0);
}
```

Figure 3 A DigitImage Class Definition

```
public class DigitImage
{
    public int width; // 28
    public int height; // 28
    public byte[][] pixels; // 0(white) - 255(black)
    public byte label; // '0' - '9'

    public DigitImage(int width, int height, byte[][] pixels, byte label)
    {
        this.width = width; this.height = height;
        this.pixels = new byte[height][];
        for (int i = 0; i < this.pixels.Length; ++i)
            this.pixels[i] = new byte[this.width];

        for (int i = 0; i < height; ++i)
            for (int j = 0; j < width; ++j)
                this.pixels[i][j] = pixels[i][j];

        this.label = label;
    }
}
```

Method LoadData reads, but doesn't use, the header information. You might want to check the four values (2051, 60000, 28, 28) to verify the file hasn't been damaged. After opening both files and reading the header integers, LoadData reads 28 x 28 = 784 consecutive pixel values from the pixel file and stores those values, then reads a single label value from the label file and combines it with the pixel values into a DigitImage object, which it then stores into the class-scope trainData array. Notice there's no explicit image ID. Each image has an implicit index ID, which is the image's zero-based position in the sequence of images.

Displaying an Image

I double-clicked on the button2 control to register its event handler. The code to display an image is shown in **Figure 5**.

The index of the image to display is fetched from the textBox4 (next image index) control, then a reference to the image is pulled from the trainImage array. You might want to add a check to make sure the image data has been loaded into memory before trying to access an image. The image is displayed in two ways, first in a visual form in the PictureBox control, and second, as hexadecimal values in the large TextBox control. A PictureBox control's Image property

Figure 4 The LoadData Method

```
public static DigitImage[] LoadData(string pixelFile, string labelFile)
{
    int numImages = 60000;
    DigitImage[] result = new DigitImage[numImages];

    byte[][] pixels = new byte[28][];
    for (int i = 0; i < pixels.Length; ++i)
        pixels[i] = new byte[28];

    FileStream ifsPixels = new FileStream(pixelFile, FileMode.Open);
    FileStream ifsLabels = new FileStream(labelFile, FileMode.Open);

    BinaryReader brImages = new BinaryReader(ifsPixels);
    BinaryReader brLabels = new BinaryReader(ifsLabels);

    int magic1 = brImages.ReadInt32(); // stored as big endian
    magic1 = ReverseBytes(magic1); // convert to Intel format
    int imageCount = brImages.ReadInt32();
    imageCount = ReverseBytes(imageCount);
    int numRows = brImages.ReadInt32();
    numRows = ReverseBytes(numRows);
    int numCols = brImages.ReadInt32();
    numCols = ReverseBytes(numCols);

    int magic2 = brLabels.ReadInt32();
    magic2 = ReverseBytes(magic2);
    int numLabels = brLabels.ReadInt32();
    numLabels = ReverseBytes(numLabels);

    for (int di = 0; di < numImages; ++di)
    {
        for (int i = 0; i < 28; ++i) // get 28x28 pixel values
        {
            for (int j = 0; j < 28; ++j) {
                byte b = brImages.ReadByte();
                pixels[i][j] = b;
            }
        }

        byte lb1 = brLabels.ReadByte(); // get the label
        DigitImage dImage = new DigitImage(28, 28, pixels, lb1);
        result[di] = dImage;
    } // Each image

    ifsPixels.Close(); brImages.Close();
    ifsLabels.Close(); brLabels.Close();

    return result;
}
```


Figure 5 Displaying an MNIST Image

```
private void button2_Click(object sender, EventArgs e)
{
    // Display 'next' image
    int nextIndex = int.Parse(textBox4.Text);
    DigitImage currImage = trainImages[nextIndex];

    int mag = int.Parse(comboBox1.SelectedItem.ToString());
    Bitmap bitMap = MakeBitmap(currImage, mag);
    pictureBox1.Image = bitMap;

    string pixelVals = PixelValues(currImage);
    textBox5.Text = pixelVals;

    textBox3.Text = textBox4.Text; // Update curr idx
    textBox4.Text = (nextIndex + 1).ToString(); // ++next index

    listBox1.Items.Add("Curr image index = " +
        textBox3.Text + " label = " + currImage.label);
}
```

can accept a Bitmap object and then render the object. Very nice! You can think of a Bitmap object as essentially an image. Note that there's a .NET Image class, but it's an abstract base class that's used to define the Bitmap class. So the key to displaying an image is to generate a Bitmap object from the program-defined DigitImage object. This is done by helper method MakeBitmap, which is listed in Figure 6.

The method isn't long but it is a bit subtle. The Bitmap constructor accepts a width and a height as integers, which for basic MNIST data will always be 28 and 28. If the magnification value is 3, then the Bitmap image will be $(28 * 3)$ by $(28 * 3) = 84$ by 84 pixels in size, and each 3-by-3 square in the Bitmap will represent one pixel of the original image.

Supplying the values for a Bitmap object is done indirectly through a Graphics object. Inside the nested loop, the current pixel value is complemented by 255 so that the resulting image will be a black/gray digit against a white background. Without complementing, the image would be a white/gray digit against a black background. To make a grayscale color, the same values for the red, green, and blue parameters are passed to the FromArgb method. An alternative is to pass the pixel value to just one of the RGB parameters to get a colored image (shades of red, green or blue) rather than a grayscale image.

The FillRectangle method paints an area of the Bitmap object. The first parameter is the color. The second and third parameters are the x and y coordinates of the upper-left corner of the rectangle. Notice that x is up-down, which corresponds to index j into the source

Figure 6 The MakeBitmap Method

```
public static Bitmap MakeBitmap(DigitImage dImage, int mag)
{
    int width = dImage.width * mag;
    int height = dImage.height * mag;
    Bitmap result = new Bitmap(width, height);
    Graphics gr = Graphics.FromImage(result);
    for (int i = 0; i < dImage.height; ++i)
    {
        for (int j = 0; j < dImage.width; ++j)
        {
            int pixelColor = 255 - dImage.pixels[i][j]; // black digits
            Color c = Color.FromArgb(pixelColor, pixelColor, pixelColor);
            SolidBrush sb = new SolidBrush(c);
            gr.FillRectangle(sb, j * mag, i * mag, mag, mag);
        }
    }
    return result;
}
```

image's pixel matrix. The fourth and fifth parameters to FillRectangle are the width and height of the rectangular area to paint, starting from the corner specified by the second and third parameters.

For example, suppose the current pixel to be displayed is at $i = 2$ and $j = 5$ in the source image, and has value = 200 (representing a dark gray). If the magnification value is set to 3, the Bitmap object will be 84-by-84 pixels in size. The FillRectangle method would start painting at $x = (5 * 3) =$ column 15 and $y = (2 * 3) =$ row 6 of the Bitmap, and paint a 3-by-3 pixel rectangle with color (55,55,55) = dark gray.

Displaying an Image's Pixel Values

If you refer back to the code in Figure 5, you'll see that helper method PixelValues is used to generate the hexadecimal representation of an image's pixel values. The method is short and simple:

```
public static string PixelValues(DigitImage dImage)
{
    string s = "";
    for (int i = 0; i < dImage.height; ++i) {
        for (int j = 0; j < dImage.width; ++j) {
            s += dImage.pixels[i][j].ToString("X2") + " ";
        }
        s += Environment.NewLine;
    }
    return s;
}
```

The method constructs one long string with embedded newline characters, using string concatenation for simplicity. When the string is placed into a TextBox control that has its Multiline property set to True, the string will be displayed as shown in Figure 1. Although hexadecimal values may be a bit more difficult to interpret than base 10 values, hexadecimal values format more nicely.

Where to from Here?

Image recognition is a problem that's conceptually simple but extremely difficult in practice. A good first step toward understanding IR is to be able to visualize the well-known MNIST data set as shown in this article. If you look at Figure 1, you'll see that any MNIST image is really nothing more than 784 values with an associated label, such as "4." So image recognition boils down to finding some function that accepts 784 values as inputs and returns, as output, 10 probabilities representing the likelihoods that the inputs mean 0 through 9, respectively.

A common approach to IR is to use some form of neural network. For example, you could create a neural network with 784 input nodes, a hidden layer of 1,000 nodes and an output layer with 10 nodes. Such a network would have a total of $(784 * 1000) + (1000 * 10) + (1000 + 10) = 795,010$ weights and bias values to determine. Even with 60,000 training images, this would be a very difficult problem. But there are several fascinating techniques you can use to help get a good image recognizer. These techniques include using a convolutional neural network and generating additional training images using elastic distortion. ■

DR. JAMES MCCAFFREY works for Microsoft Research in Redmond, Wash. He has worked on several Microsoft products including Internet Explorer and Bing. McCaffrey can be reached at jammc@microsoft.com.

THANKS to the following technical expert for reviewing this article:
Wolf Kienzle (Microsoft Research)



IT AND DEVELOPER TRAINING THAT'S OUT OF THIS WORLD.

Live! 360 brings together five conferences, and the brightest minds in IT and Dev, to explore leading edge technologies and conquer current ones. These co-located events will incorporate knowledge transfer and networking, along with out-of-this-world education and training, as you create your own custom conference, mixing and matching sessions and workshops to best suit your needs.

5 GREAT CONFERENCES. 1 GREAT PRICE.

Visual Studio **LIVE!**

EXPERT SOLUTIONS FOR .NET DEVELOPERS

Calling all .NET Developers: The Code is Strong with This One; Experience unbiased and practical training on the Microsoft Platform.

SharePoint **LIVE!**

TRAINING FOR COLLABORATION

USS Collaborate: Let's Work Together. Climb aboard to work through your most pressing SharePoint projects.

SQL Server **LIVE!**

TRAINING FOR DBAs AND IT PROS

SQL Server Live! will leave you with the skills needed to Conquer Planet Data, whether you are a DBA, developer, IT Pro, or Analyst.

ModernApps **LIVE!**

MOBILE, CROSS-DEVICE & CLOUD DEVELOPMENT

Master the Modern Apps Landscape and learn how to architect, design and build a complete Modern Application from start to finish.

TECHMENTOR

IN-DEPTH TRAINING FOR IT PROS

Join us for an IT Training Odyssey, focused entirely on making your datacenter more modern, more capable, and more manageable.

Five events, 35+ tracks, and literally hundreds of sessions to choose from – mix and match sessions to create your own, custom event line-up – it's like no other conference available today.



REGISTER BY AUGUST 6 AND SAVE \$500!

Use promo code L360JUN2



Scan the QR code to register or for more event details.

CONNECT WITH LIVE!360



twitter.com/@live360events



facebook.com – Search "Live 360"



linkedin.com – Join the "Live 360" group!



The Canvas and the Camera

About 15 years ago, English artist David Hockney began developing a theory about Renaissance art that turned out to be quite controversial. Hockney marveled at how Old Masters such as van Eyck, Velázquez, and Leonardo were able to render the visual world on canvas with astounding accuracy in perspective and shading. He became convinced that this accuracy was possible only with the help of optical tools made from lenses and mirrors, such as the camera obscura and camera lucida. These devices flatten the 3D scene and bring it closer to the artist for reproduction. Hockney published his thesis in a gorgeous and persuasive book entitled, “Secret Knowledge: Rediscovering the Lost Techniques of the Old Masters” (Viking, 2001).

More recently, inventor Tim Jenison (founder of NewTek, the company that developed Video Toaster and LightWave 3D), became obsessed with using optical tools to recreate Jan Vermeer’s 350-year-old painting “The Music Lesson.” He built a room similar to the original, decorated it with furniture and prop reproductions, recruited live models (including his daughter), and used a simple optical tool of his own invention to paint the scene. The jaw-dropping results are chronicled in the fascinating documentary, “Tim’s Vermeer.”

Why is it necessary to use optical devices—or, in these days, a simple camera—to capture 3D scenes on 2D surfaces accurately? Much of what we think we see in the real world is constructed in the brain from relatively sparse visual information. We think we see a whole real-world scene in one big panorama, but at any point in time, we’re really focused only on a small detail of it. It’s virtually impossible to piece together these separate visual fragments into a composite painting that resembles the real world. It’s a lot easier when the canvas is supplemented with a mechanism much like a camera—which itself mimics the optics of the human eye, but without the eye’s deficiencies.

A similar process occurs in computer graphics: When rendering 2D graphics, a metaphorical canvas works just fine. The canvas is a drawing surface, and in its most obvious form, it corresponds directly to the rows and columns of pixels that comprise the video display.

But when moving from 2D to 3D, the metaphorical canvas needs to be supplemented with a metaphorical camera. Like a real-world camera, this metaphorical camera captures objects in 3D space and flattens them onto a surface that can then be transferred to the canvas.

Camera Characteristics

A real-world camera can be described with several characteristics that are easily converted into the mathematical descriptions required by 3D graphics. A camera has a particular position in space, which can be represented as a 3D point.

The camera is also aimed in a particular direction: a 3D vector. You can calculate this vector by obtaining the position of an object the camera is pointed directly at, and subtracting the camera position.

If you keep a camera fixed at a particular position and pointed in a particular direction, you still have the freedom to tilt the camera side to side, and actually rotate it 360 degrees. This means that another vector is required to indicate “up” relative to the camera. This vector is perpendicular to the camera direction.

After establishing how the camera is positioned in space, you get to fiddle around with the knobs on the camera.

Today’s cameras are often capable of zoom: You can adjust the camera lens from a wide angle that encompasses a big scene to a telephoto view that narrows for a close up. The difference is based on the distance between the plane that captures the image, and the focal point, which is a single point through which the light passes, as shown in **Figure 1**. The distance between the focal plane and the focal point is called the focal length.

Much of what we think we see in the real world is constructed in the brain from relatively sparse visual information.

The sizes of the focal plane and the focal length imply an angular field of view emanating from the focal point. Telephoto views (more correctly called “long focus”) are generally associated with a field of view less than 35 degrees, while a wide angle is greater than 65 degrees, with more normal fields of view in between.

Computer graphics add a characteristic of the camera not possible in real life: In 3D graphics programming you often have a choice between cameras that achieve either *perspective* or *orthographic* projection. Perspective is like real life: Objects further from the camera appear to be smaller because the field of view encompasses a greater range further from the focal point.

Code download available at msdn.microsoft.com/magazine/msdnmag0614.



YOUR .NET Resources



Visual Studio[®]
MAGAZINE

Visual Studio[®] **LIVE!**
EXPERT SOLUTIONS FOR .NET DEVELOPERS

ONLINE | NEWSLETTERS | PRINT | CONFERENCES

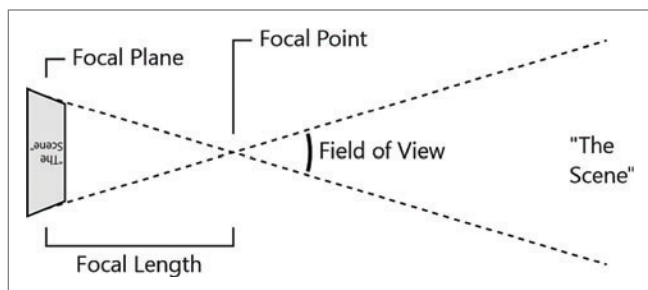


Figure 1 The Focal Plane, Focal Point and Focal Length

With orthographic projection, that's not the case. Everything is rendered in a size relative to the object's actual size, regardless of its distance from the camera. Mathematically, this is the simpler of the two projections, and is most appropriate for technical and architectural drawings.

The Camera Transforms

In 3D graphics programming, cameras are mathematical constructs. The camera consists of two matrix transforms much like those that manipulate objects in 3D space. The two camera transforms are called View and Projection.

The View matrix effectively positions and orients the camera in 3D space; the Projection matrix describes what the camera "sees" and how it sees it. These camera transforms are applied after all the other matrix transforms are used to position objects in 3D space, often called "world space." Following all the other transforms, first the View transform is applied, and finally the Projection transform.

We think we see a whole
real-world scene in one big
panorama, but at any point in
time, we're really focused only on
a small detail of it.

In DirectX programming—whether Direct3D or the exploration of 3D concepts in Direct2D—it's easiest to construct these matrix transforms using the DirectX Math Library, the collection of functions in the DirectX namespace that begin with the letters XM and use the XMVECTOR and XMMATRIX data types. These two data types are proxies for CPU registers, so these functions are often quite speedy.

Four functions are available to calculate the View matrix:

- XMMatrixLookAtRH (EyePosition, FocusPosition, UpDirection)
- XMMatrixLookAtLH (EyePosition, FocusPosition, UpDirection)
- XMMatrixLookToRH (EyePosition, EyeDirection, UpDirection)
- XMMatrixLookToLH (EyePosition, EyeDirection, UpDirection)

The function arguments include the word "Eye" but the documentation uses the word "camera."

The LH and RH abbreviations stand for left-hand and right-hand. I'll be assuming a left-hand coordinate system for these examples. If

you point the index finger of your left hand in the direction of the positive X axis, and your middle finger in the direction of positive Y, your thumb will point to positive Z. If the positive X axis goes right, and positive Y goes up (a common orientation in 3D programming) then the -Z axis comes out of the screen.

All four functions require three objects of type XMVECTOR and return an object of type XMMATRIX. In all four functions, two of these arguments indicate the camera position (labeled as EyePosition in the function template) and the UpDirection. The LookAt functions include a FocusPosition argument—a position the camera is pointed at—while the LookTo functions have an EyeDirection, which is a vector. It's just a simple calculation to convert from one form to the other.

For example, suppose you want to position the camera at the point (0, 0, -100), pointing toward the origin (and, hence, in the direction of the positive Z axis), with the top of the camera pointing up. You can call either

```
XMMATRIX view =
    XMMatrixLookAtLH(XMVectorSet(0, 0, -100, 0),
                     XMVectorSet(0, 0, 0, 0),
                     XMVectorSet(0, 1, 0, 0));
```

or

```
XMMATRIX view =
    XMMatrixLookToLH(XMVectorSet(0, 0, -100, 0),
                     XMVectorSet(0, 0, 1, 0),
                     XMVectorSet(0, 1, 0, 0));
```

In either case, the function creates this View matrix:

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 100 & 1 \end{vmatrix}$$

This particular View matrix simply shifts the entire 3D scene 100 units in the direction of the positive Z axis. Many View matrices will also involve rotations of various sorts, but no scaling. After the View transform is applied to the 3D scene, the camera can be assumed to be positioned at the origin (with the top of the camera pointed in the direction of positive Y) and pointed in the positive Z direction (for a left-hand system) or the -Z direction (right-hand). This orientation allows the Projection transform to be much simpler than it would be otherwise.

The Projection Conventions

In this column's previous forays into 3D programming within Direct2D, I've converted objects from 3D space to the 2D video display simply by ignoring the Z coordinate.

It's time to convert from 3D to 2D in a more professional manner using conventions that are encapsulated in the standard camera Projection matrices. The standard conversion of 3D to 2D actually occurs in two stages: first from 3D coordinates to normalized 3D coordinates, and then to 2D coordinates. The Projection transform specified by the program controls the first conversion. In Direct3D programming, the second conversion is usually performed automatically by the rendering system. A program that uses Direct2D to display 3D graphics must perform this second conversion itself.

The purpose of the Projection transform is in part to normalize all the 3D coordinates in the scene. This normalization defines what objects are visible in the final rendering and which are excluded. Following this normalization, the final rendered scene encompasses X coordinates ranging from -1 at the left and 1 at the right, Y coordinates ranging from -1 on the bottom to 1 at the top, and Z coordinates ranging from 0 (closest to the camera) to 1 (furthest from the camera). The Z coordinates are also used to determine what objects obscure other objects relative to the viewer.

Everything not in this space is discarded, and then the normalized X and Y coordinates are mapped to the width and height of the display surface, while the Z coordinates are ignored.

To normalize the Z coordinates, the functions that compute a Projection matrix always require arguments of type float named NearZ and FarZ that indicate a distance from the camera along the Z axis. These two distances are converted to normalized Z coordinates of 0 and 1, respectively.

But when moving from 2D to 3D,
the metaphorical canvas needs
to be supplemented with a
metaphorical camera.

This is somewhat counterintuitive because it implies there's an area of 3D space that's too close to the camera to be visible, and another area that's too far away. But for practical reasons it is necessary to limit depth in this way. Everything behind the camera must be eliminated, for example, and objects too close to the camera would obscure everything else. If Z coordinates out to infinity were allowed, the resolution of floating point numbers would be taxed when determining what objects overlap others.

Because the camera View matrix accounts for possible translation and rotation of the camera, the Projection matrix is always based on a camera located at the origin and pointing along the Z axis. I'll be using left-hand coordinates for these examples, which means the camera is pointed in the direction of the positive Z axis. Left-hand coordinates are a little simpler when dealing with Projection transforms because NearZ and FarZ are equal to coordinates along the positive Z axis rather than the -Z axis.

The DirectX Math Library defines 10 functions for calculating the Projection matrix—four for orthographic projections and six for perspective projections, half for left-hand coordinates and half for right-hand.

In the XMMatrixOrthographicRH and LH functions, you specify ViewWidth and ViewHeight along with NearZ and FarZ. **Figure 2** is a view looking down on the 3D coordinate system from a location on the positive Y axis. This figure shows how these arguments define a cuboid in a left-hand coordinate system viewable to an eyeball at the origin.

Often, the ratio of ViewWidth to ViewHeight is the same as the aspect ratio of the display used for rendering. The projection

transform scales everything from -ViewWidth / 2 to ViewWidth / 2 to the range -1 to 1, and later those normalized coordinates are scaled by half the pixel width of the display surface for rendering. The calculation is similar for ViewHeight.

Here's a call to XMMatrixOrthographicLH with ViewWidth and ViewHeight set to 40 and 20, and NearZ and FarZ set to 50 and 100, which matches the diagram assuming tick marks every 10 units:

```
XMMATRIX orthographic =  
    XMMatrixOrthographicLH(40, 20, 50, 100);
```

This results in the following matrix:

$$\begin{bmatrix} 0.05 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0.02 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

The transform formulas are:

$$\begin{aligned} x' &= 0.05x \\ y' &= 0.1y \\ z' &= 0.02z - 1 \end{aligned}$$

You can see that x values of -20 and 20 are transformed to -1 and 1, respectively, and that y values of -10 and 10 are transformed to -1 and 1, respectively. A Z value of 50 is transformed to 0, and a Z value of 100 is transformed to 1.

Two additional Orthographic functions contain the words OffCenter and let you specify left, right, top, and bottom coordinates rather than widths and heights.

The XMMatrixPerspectiveRH and LH functions have the same arguments as XMMatrixOrthographicRH and LH, but define a four-sided frustum (like a pyramid with its top cut off) as shown in **Figure 3**.

The ViewWidth and ViewHeight arguments in the transform functions control the width and height of the frustum at NearZ, but the width and height at FarZ is proportionally larger based on the ratio of FarZ to NearZ. This diagram also demonstrates how a greater range of x and y coordinates further from the camera are mapped into the same space (and, hence, are made smaller) as the x and y coordinates nearer the camera.

Here's a call to XMMatrixPerspectiveLH with the same arguments I used for XMMatrixOrthographicLH:

```
XMMATRIX perspective =  
    XMMatrixPerspectiveLH(40, 20, 50, 100);
```

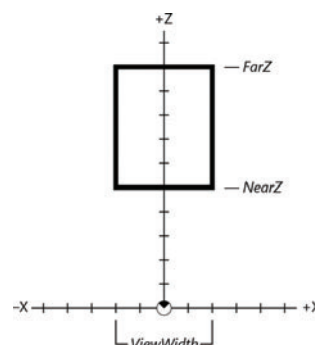


Figure 2 A Top View of an Orthographic Transform

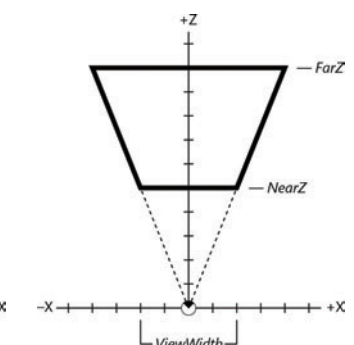


Figure 3 A Top View of a Perspective Transform

The matrix created from this call is:

$$\begin{bmatrix} 2.5 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & -100 & 0 \end{bmatrix}$$

Notice that the fourth column indicates a non-affine transform! In an affine transform, the m14, m24, and m34 values are 0, and m44 is 1. Here, m34 is 1 and m44 is 0.

This is how perspective is achieved in 3D programming environments, so let's look at the transform multiplication in detail:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \times \begin{bmatrix} 2.5 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & -100 & 0 \end{bmatrix} = \begin{bmatrix} x' & y' & z' & w' \end{bmatrix}$$

The matrix multiplication results in the following transform formulas:

$$\begin{aligned} x' &= 2.5x \\ y' &= 5y \\ z' &= 2z - 100 \\ w' &= z \end{aligned}$$

Notice the w' value. As I discussed in the April installment of this column, the use of w coordinates in 3D transforms ostensibly accommodates translation, but it also brings in the mathematics

of homogenous (or projective) coordinates. Affine transforms always take place in the 3D subset of 4D space where w equals 1, but this non-affine transform has moved coordinates out of that 3D space. The coordinates must be moved back into that 3D space by dividing them all by w' . The transform formulas that incorporate this necessary adjustment are instead:

$$\begin{aligned} x' &= \frac{2.5x}{z} \\ y' &= \frac{5y}{z} \\ z' &= 2 - \frac{100}{z} \\ w' &= \frac{z}{z} = 1 \end{aligned}$$

When z equals NearZ or 50, the transform formulas are the same as for the orthographic projection:

$$\begin{aligned} x' &= 0.05x \\ y' &= 0.1y \\ z' &= 0 \end{aligned}$$

Values of x from -20 to 20 are transformed into x' values from -1 to 1, for example.

For other values of z , the transform formulas are different, and when z equals FarZ or 100, they look like this:

$$\begin{aligned} x' &= 0.025x \\ y' &= 0.05y \\ z' &= 1 \end{aligned}$$

At this distance from the camera, values of x from -40 to 40 are transformed into x' values from -1 to 1. A larger range of x and y values at FarF occupy the same visual field as a smaller range at NearZ.

As with the orthographic functions, two additional functions have the words OffCenter in the function names, and let you set left, right, top, and bottom coordinates rather than widths and heights.

The `XMMatrixPerspectiveFovRH` and `LH` functions let you specify an angular field of view (FOV) rather than a width and height. This field of view is likely different along the X and Y axes. You need to specify it along the Y axis, and also provide a ratio of width to height.

To create a Perspective matrix consistent with the preceding example, the field of view can be calculated with the `atan2` function, with a y argument equal to half the height at NearZ, and the x argument equal to NearZ, and then doubling the result:

```
float angleY = 2 * atan2(10.0f, 50.0f);
```

The second argument is the ratio of width to height, or 2 in this example:

```
XMMATRIX perspective =
    XMMatrixPerspectiveFovLH(angleY, 2, 50, 100);
```

This call results in a Perspective matrix identical to the one just created with `XMMatrixPerspectiveLH`.

A Circle of Text

In February's installment of this column, I demonstrated how to create a 3D circle of text, and animate its rotation. However, I used `Direct2D` geometries for that exercise and encountered very poor performance.

Figure 4 The View and Projection Matrices in `TessellatedText3D`

```
void TessellatedText3DRenderer::Update(DX::StepTimer const& timer)
{
    ...

    // Calculate camera view matrix
    XMVECTOR eyePosition = XMVectorSet(0, 0, -2 * m_sourceRadius, 0);
    XMVECTOR focusPosition = XMVectorSet(0, 0, 0, 0);
    XMVECTOR upDirection = XMVectorSet(0, 1, 0, 0);

    XMMATRIX viewMatrix = XMMatrixLookAtLH(eyePosition,
                                           focusPosition,
                                           upDirection);

    // Calculate camera projection matrix
    float width = 1.5f * m_sourceRadius;
    float nearZ = 1 * m_sourceRadius;
    float farZ = nearZ + 2 * m_sourceRadius;
    XMMATRIX projMatrix = XMMatrixPerspectiveLH(width,
                                                width,
                                                nearZ,
                                                farZ);

    // Calculate composite matrix
    XMMATRIX matrix = rotateMatrix * tiltMatrix *
                     viewMatrix * projMatrix;

    // Apply composite transform to all 3D triangles
    XMVECTOR3TransformCoordStream(
        (XMFLAOT3 *) m_dstTriangles3D.data(),
        sizeof(XMFLAOT3),
        (XMFLAOT3 *) m_srcTriangles3D.data(),
        sizeof(XMFLAOT3),
        3 * m_srcTriangles3D.size(),
        matrix);

    ...
}
```

In the March column I demonstrated how to tessellate text into a collection of triangles, which seemed to have considerably better performance than geometries. In the May column, I demonstrated how to use triangles to create and render 3D objects.

It's time to put these different techniques together. The downloadable source code for this column includes a Windows 8 Direct2D project called *TessellatedText3D*. For this program I defined a 3D triangle using *XMFLOAT3* objects:

```
struct Triangle3D
{
    DirectX::XMFLOAT3 point1;
    DirectX::XMFLOAT3 point2;
    DirectX::XMFLOAT3 point3;
};
```

The constructor of the *TessellatedText3DRenderer* class loads in a font file, creates a font face, and generates an *ID2D1PathGeometry* object from the *GetGlyphRunOutline* method using an arbitrary font size of 100. This geometry is then converted into a collection of triangles using the *Tessellate* method with a custom tessellation sink. With the particular font, font size and glyph indices I specified, *Tessellate* generates 1,741 triangles.

The 2D triangles are then converted into 3D triangles by wrapping the text in a circle. Based on that arbitrary font size of 100, this circle happens to have a radius of about 200 (stored in *m_sourceRadius*), and the circle is centered on the 3D origin.

In the *Update* method of the *TessellatedText3DRenderer* class, the *XMMatrixRotationY* and *XMMatrixRotationX* functions provide two transforms to animate a rotation of the text around the X and Y axes. These are stored in *XMMATRIX* objects named *rotateMatrix* and *tiltMatrix*, respectively.

The *Update* method then continues with the code shown in **Figure 4**. This code calculates the View and Projection matrices. The View matrix sets the camera position on the -Z axis based on the circle radius, so the camera is a radius length outside the circular text but pointed towards the center.

With arguments also based on the circle radius, the code continues by calculating a Projection matrix, and then multiplies all the matrices together. The *XMVector3TransformCoordStream* uses parallel processing to apply this transform to an array of *XMFLOAT3* objects (actually an array of *Triangle3D* objects), automatically performing the division by *w*.

The *Update* method continues beyond what's shown in **Figure 4** by converting those transformed 3D coordinates to 2D using a scaling factor based on half the width of the video display, and

ignoring the z coordinates. The *Update* method also uses the 3D vertices of each triangle to calculate a cross product, which is the surface normal—a vector perpendicular to the surface. The 2D triangles are then divided into two groups based on the z coordinate of the surface normal. If the z coordinate is negative, the triangle is facing toward the viewer, and if it's positive, the triangle is facing away. The *Update* method concludes by creating two *ID2D1Mesh* objects based on the two groups of 2D triangles.

Perspective is like real life: Objects further from the camera appear to be smaller because the field of view encompasses a greater range further from the focal point.

The *Render* method then displays the mesh of rear triangles with a gray brush and the front mesh with a blue brush. The result is shown in **Figure 5**.

As you can see, the front-facing triangles closer to the viewer are much larger than the back-facing triangles further away. This program has none of the performance problems encountered with rendering geometries.

Shading with Light?

In the May installment of this column, I took light into account when displaying 3D objects. The program assumes light comes from a particular direction and it calculates different shading for the sides of the solid figures based on the angle that the light strikes each surface.

Is something similar possible with this program?

In theory, yes. But my experiments revealed some serious performance problems. Rather than creating two *ID2D1Mesh* objects in the *Update* method, a program implementing shading of the text needs to render each triangle with a different color, and that requires 1,741 different *ID2D1Mesh* objects recreated in each *Update* call, and 1,741 corresponding *ID2D1SolidColorBrush* objects. This slowed the animation down to approximately one frame per second.

What's worse, the visuals weren't satisfactory. Each triangle gets a different discrete solid color based on its angle to the light source, but the boundaries between these discrete colors became visible! Triangles used to render 3D objects must more properly be colored with a gradient between the three vertices, but such a gradient is not supported by the interfaces that derive from *ID2D1Brush*.

This means I must dig even deeper into Direct2D and get access to the same shading tools that Direct3D programmers use. ■



Figure 5 The *TessellatedText3D* Display

CHARLES PETZOLD is a longtime contributor to MSDN Magazine and the author of *Programming Windows, 6th Edition* (Microsoft Press, 2013), a book about writing applications for Windows 8. His Web site is charlespetzold.com.

THANKS to the following Microsoft technical expert for reviewing this article: Doug Erickson



VB6 and the Art of the Knuckleball

Two years ago in this column I wrote about the astonishing longevity of Visual Basic 6 (msdn.microsoft.com/magazine/jj133828). I got more comments on that column than anything I've ever written—145 at last count and still arriving as I write these words. Clearly, I touched some very strong feelings.

I've often wondered at the staying power of Visual Basic 6 (VB6). A student of mine once called VB6 the “unkillable cockroach” of the Windows ecosystem. So when I heard a Boston sports announcer comment on the 17-year career of a beloved Red Sox knuckleball pitcher and say, “After the apocalypse, all we'll have left is cockroaches and Tim Wakefield,” it hit me: VB6 is like a knuckleball.

The knuckleball is a tricky pitch in American baseball. Unlike a fastball, which speeds past batters before they can swing, the slower knuckleball confounds opponents by fluttering and dancing randomly through the air. The pitcher throws the ball with almost no spin, holding it in his fingernails and pushing it forward (see bit.ly/1jDB3a4). The motion puts much less stress on the arm than other pitches. The few pitchers to master the knuckleball have enjoyed much longer careers than their fireball-throwing counterparts. VB6 is like that. Its simplicity subjects programmers to less stress, resulting in longer careers.

Knuckleball pitchers and VB6 programmers share another notable trait—they stick together.

The analogy gets better. Many, perhaps most, fastball pitchers consider the knuckleball unsporting, a circus stunt unworthy of a “real pitcher.” Check out the comments from my first VB6 column, and you can see this meme at work: “VB6 is a toy, not a real programming language, and people who use it are not real programmers.” I once said the same thing myself, back in my C++ COM days, for which I now apologize.

You can rarely complete a major project using only VB6. While many programming operations are far easier in VB6 than in raw C++, other operations (background threads, say) are essentially impossible. My rule of thumb for VB6 development has always been, “If you can't do it in 10 minutes, you can't do it at all.” I always advise VB6 development teams that they need one C++ programmer to write COM objects to accomplish the few operations that Visual Basic can't.

Similarly, a knuckleball pitcher needs to throw fastballs about 20 percent of the time to keep hitters honest, especially with runners on third base, where a passed ball can score a run. Catchers often struggle with the hard-to-catch knuckleball (as Bob Uecker famously observed, “The way to catch a knuckleball is to wait until it stops rolling and then pick it up.”).

I've never met anyone who majored in computer science intending to become a VB6 programmer. Most of the VB6 guys I know started as subject matter experts, learning Visual Basic as the easiest way to start computerizing their problem domain knowledge. Similarly, few pitchers start their careers throwing the knuckleball. Blogger Peter Duffy writes that most start throwing it when they can't perform as conventional pitchers, “out of desperation; a way to salvage what is left of a lifelong dream.” (bit.ly/1lazW5S)

When a knuckleball works, it's unhittable. But if the pitcher's fingernail slips and the pitch doesn't dance, he's toast. The ball sort of hangs slowly in front of the batter and gets smacked out of the park. Likewise, VB6 is great when it works. But it's hard to debug when it seriously breaks because you can't see under the hood (“0x80014005 — Unknown Error”).

Knuckleball pitchers and VB6 programmers share another notable trait—they stick together. The documentary film “Knuckleball!” (bit.ly/1jfv5JU) shows retired knuckleballers—Jim Bouton, Charlie Hough, Phil Niekro—helping Tim Wakefield to mentor R.A. Dickey, today's only active big league knuckleballer. It worked: Dickey won the Cy Young Award as the National League's best pitcher in 2012. Now Dickey is helping Red Sox AAA knuckleballer Steven Wright try to break into the major leagues this year. I find the same collegiality in the VB6 community, especially now that Microsoft isn't driving it.

I recently taught .NET programming to some scientists, who needed to write programs for controlling their instruments. Despite my best efforts, the complexity of Windows Presentation Foundation, Model-View-ViewModel and the Microsoft .NET Framework severely distracted them from their science. I wish, and they wish, that a simpler tool existed for .NET development. I wish I could have taught them to throw a knuckleball. ■

DAVID S. PLATT teaches programming .NET at Harvard University Extension School and at companies all over the world. He's the author of 11 programming books, including “Why Software Sucks” (Addison-Wesley Professional, 2006) and “Introducing Microsoft .NET” (Microsoft Press, 2002). Microsoft named him a Software Legend in 2002. He wonders whether he should tape down two of his daughter's fingers so she learns how to count in octal. You can contact him at rollthunder.com.

.NET TOOLS FOR DEV PROS

Whether you're building the most modern touch-enabled apps or maintaining and updating legacy applications, our flagship product, Studio Enterprise, helps to deliver rich, responsive, desktop and web apps on time and under budget.

DataGrids

STUDIO
ENTERPRISE

Reporting

Data
Visualization

Touch

HTML5

New 2014 v1 Release!

ComponentOne®
a division of GrapeCity®

DOWNLOAD YOUR FREE TRIAL
► componentone.com/se

© 2014 GrapeCity, Inc. All rights reserved. All other product and brand names are trademarks, and/or registered trademarks of their respective holders.

HOBBYIST LICENSE OFFER

ONLY \$1

\$599
VALUE!

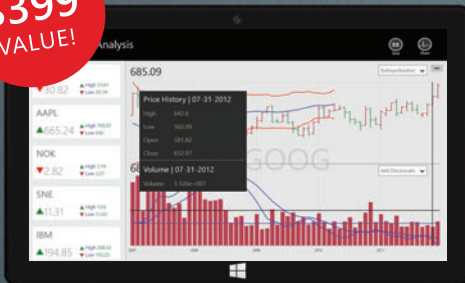


JavaScript

syncfusion.com/jshobbyistMSDN



\$399
VALUE!



WinRT

syncfusion.com/winrthobbyistMSDN



\$99
VALUE!



Windows Phone

syncfusion.com/winphonehobbyistMSDN



- ✓ 30+ controls: charts, grids, gauges, and more
- ✓ One year of support and updates
- ✓ Individual developers qualify

Buy online and get
your license today!